National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

Energy-Optimal Control of a Bilinear Multizone Cooling System

BY

Nan Li ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

DEPARTMENT OF ELECTRICAL ENGINEERING

EDMONTON, ALBERTA
SPRING 1995

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **Nan Li**

TITLE OF THESIS: **Energy-Optimal Control of a Bilinear Multizone Cooling System**

DEGREE: **Master of Science**

YEAR THIS DEGREE GRANTED: **1995**

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Nan Li

#15-8616, 108 Street
Edmonton, Alberta
Canada

December 19, 1994

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-01623-4

Canadä

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Energy-Optimal Control of a Bilinear Multizone Cooling System** submitted by **Nan Li** in partial fulfillment of the requirements of the degree of **Master of Science**.

R. E. Rink
Dr. R.E Rink (Supervisor)

Jack W. Macki
Dr. J. Macki

Keith A. Stromsmoe
Dr. K. Stromsmoe

November 25, 1994

Abstract

Efficient operation of multizone fan-coil cooling systems with chilled-water storage is studied in this thesis. First the system is considered as consisting of two environmental zones plus storage, the model of the system is bilinear, which is linearized through the use of state feedback. For this case, when electrical energy is priced at a discount during off-peak hours, the controller algorithms which provide close regulation of the zone temperatures at minimum energy cost are obtained through the method of state increment dynamic programming, augmented with a label-tracing procedure to identify the periodic but unknown initial/terminal state. The results lead to several observations concerning economical operating strategies, and the effect on electrical energy purchase or consumption is significant. But treating the system as two environmental zones is not always realistic, because a large office building has many rooms which have their own environmental conditions and have different temperature demands. The two zone cooling system should be augmented to multizone. Since the augmentation is limited by the high-speed memory of the computer, a disaggregation/aggregation method is developed and the solution by this method, which is suboptimal, is evaluated by comparing with truly optimal one. Another method is to compute optimal control only in a region in which the optimal trajectories of interest are expected to lie, then computations are performed only in those blocks contained in it and hence a considerable reduction in both computing time and memory requirement can be obtained. Softwares are designed in C on Unix for both these methods.

Acknowledgements

I would like to express my heartfelt gratitude to my supervisor Dr. R.E. Rink who has given me patient guidance, constant encouragement and financial support throughout this research work, from him, I learn not only knowledge but also how to be a human being. I deeply believe his virtues will affect my whole life.

Technical assistance from Kees and Norman has been an immense contribution and is greatly appreciated. Without them, the computer program work could not have be done.

Special appreciation should be extented to Dr. R.P.W. Lawson for his concerns and support. I also like to thank the department of Electrical Engineering for providing the opportunity and all kinds of support.

Finally, a well-deserved expression of appreciation goes to my husband and family members, and to all my teachers for their invaluable contributions.

Contents

1 Introduction **1**

1.1 Cooling System with Chilled-Water Storage 2

1.2 Research Objectives 3

1.3 Thesis Organization 5

2 The Principles of Dynamic Programming **6**

2.1 Introduction 6

2.2 Dynamic Programming Fundamentals 7

2.2.1 Problem Formulation 7

2.2.2 Bellman's Principle Of Optimality 10

2.2.3 Derivation Of The Iterative Functional Equation 11

2.3 Conventional Dynamic Programming Computational Procedure . 12

2.3.1 Computational Procedure 12

2.3.2 Computational Requirements 17

2.4 State Increment Dynamic Programming 20

2.4.1 Basic Concepts of State Increment Dynamic Programming 20

2.4.2 Computation Procedure Within a Block in the General Case 24

2.4.3 Procedure for Processing Blocks 25

2.4.4 Computational Requirement 27

3 Efficient Operation of a Bilinear Two-zone Cooling System with Chilled-Water Storage 29

3.1 Introduction 29

3.2 Problem Formulation 31

3.3 Linearization by State Feedback 36

3.3.1 Optimization by State Increment Dynamic Programming . 38

3.3.2 Computation Requirement 41

3.3.3 Results and Conclusions 42

4 Aggregation and Disaggregation Method and Block by Block Approach 48

4.1 Multizone Cooling Systems With Many Zones 48

4.2 Computational Requirement 49

4.3 Aggregation and Disaggregation Method 50

4.3.1 Principles of the Aggregation and Disaggregation Method . 50

4.3.2 Optimization by State Increment Dynamic Programming . 56

4.3.3 Application of Aggregation and Disaggregation Method . . 58

4.4 Block by Block Method 72

4.4.1 Principles of The Block by Block Method 72

4.5 Application of Block by Block Method 75

5 Conclusions **84**

A Source Code **89**

A.1 Source Code for Two-Zone Cooling System 90

A.2 Source Code for Eight-Zone Cooling System 99

A.3 Source Code for Four-Zone Cooling System 134

Chapter 1

Introduction

Although many methods of heating and cooling have been used for thousands of years (hand fanning, living in caves, wood and oil fires), Leonardo da Vinci was probably the first inventor of an automatic cooling system. In 1500 A.D., he built a water-driven fan to ventilate a suite of rooms for the wife of a patron. However, keeping cool was not easy for ordinary people until recently. The first cooling-fan installation was made in a theater in 1922. Since that time, virtually every type of building has been air conditioned, from giant skyscrapers to small homes. Air conditioning (cooling) becomes an important part of people's daily life. The problem of improving or optimizing the control of heating and/or cooling system has received some attention in recent years, since the practical solutions of this problem have economic significance.

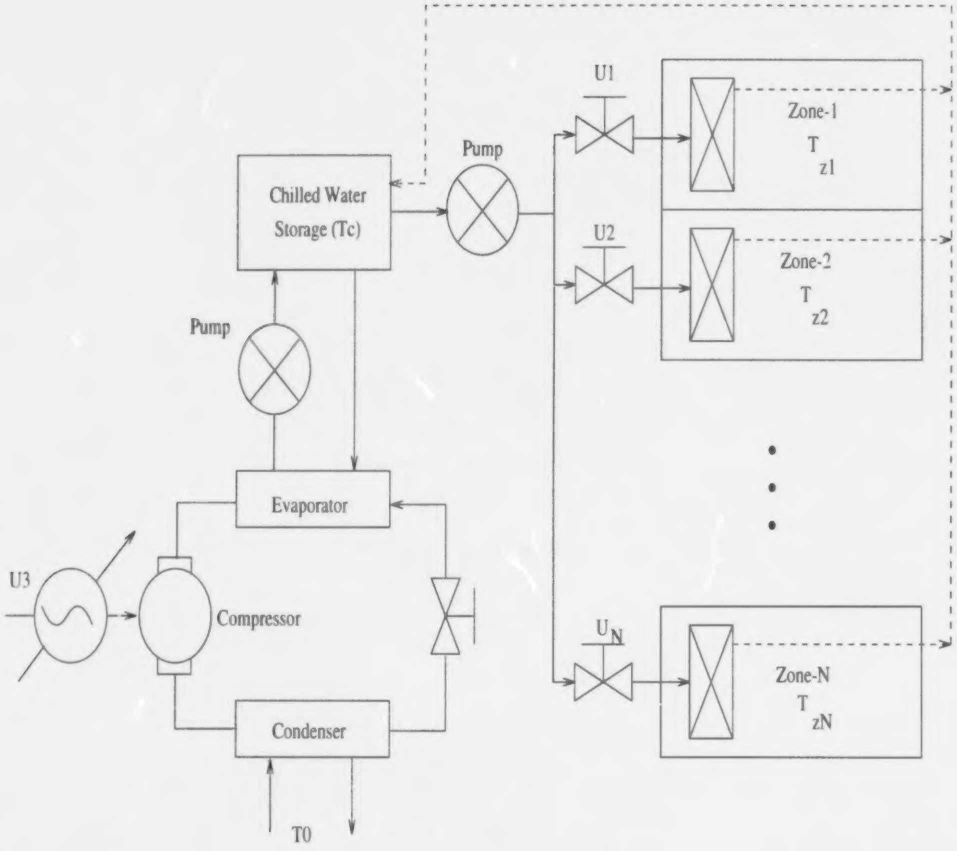

U1
Zone-1
T_{z1}
Pump
Chilled Water
Storage (Tc)
U2
Zone-2
T_{z2}
Pump
Evaporator
U3
Compressor
U_N
Zone-N
T_{zN}
Condenser
T0

Figure 1.1: Schematic diagram of a multizone cooling system.

1.1 Cooling System with Chilled-Water Storage

The cooling system with chilled-water storage mainly consists of three parts. The compression-cycle refrigeration unit (chiller); storage tank; and cooling zones. A schematic diagram of the system is shown in Figure 1.1[1].

The prime function of the compression-cycle refrigeration unit is moving heat from chilled storage[10].

Each zone of the system has its own fan-coil unit which consists of a cooling coil, a circulating fan and supply and return pipes. The low temperature water of the storage tank is pumped to supply pipes and carried to the coil, and then comes back to the storage tank at a higher temperature through the return pipes. Electric energy is consumed by the circulation pump.

This system sounds simple but its optimal control is challenging, analytically, since the simplest reasonable models for the system dynamics are nonlinear. The cost function, moreover, is nonquadratic and the state and control variables are bounded. Serveral related optimization problems, in the context of HVAC systems for zone heating/cooling , have been studied[1–6], and several successful suboptimal[5–6], and optimal[1–4] control synthesis procedures have been found. In [1] it was shown that an optimization method , state increment dynamic programming , simplified through the use of the state feedback linearization and decoupling, and augmented with a lable-tracing procedure to identify the periodic but unknown initial/terminal state, provides a reliable way of solving such problems.

1.2 Research Objectives

The problem under consideration is that of finding the minimum-cost strategy for operating a multizone cooling system that might be typical of an office building divided into a number of zones. Of particular interest is the case where the storage capacity is substantial, for then the chiller can be mainly operated at times when zone cooling loads are low, e.g. to take advantage of off-peak electrical energy that may be offered at a discounted price by the utility company.

Under the assumption that the zone cooling-load profiles are going to be periodic, with 24-hour period, the method of state increment dynamic programming, augmented with a label-tracing procedure[1], can be used to project the optimal periodic temperature trajectories for the chilled storage and zones such that the cost of energy consumption is minimum.

While reliable, this method was found to be practical only for systems with a small number of zones, e.g. four or fewer, since the "curse of dimensionality", common to all applications of dynamic programming, makes the computational workload excessive when the number of state variables (zone temperatures plus storage temperature) exceeds five or so. In this respect, the main objective of the research undertaken for this thesis was to extent the application of dynamic programming methods to cooling systems with many zones. This involved the study of:

- Dynamic programming, comparing the conventional dynamic programming method with the state increment dynamic programming method.
- Efficient operation of two-zone systems with chilled-water storage, via using the state increment dynamic programming method directly.
- Aggregation/disaggregation sequential approach, developed for extending the dynamic programming method to multizone systems of much larger size. The suboptimal solution obtained by this technique is evaluated by comparing with the optimal one.
- For the same purpose as above, a block by block method is designed, which provides optimal solutions.

Both new approaches are practical and reliable. As examples, the results of aggregation/disaggregation sequential approach as applied to eight zones and block by block method to four zones are provided and discussed.

1.3 Thesis Organization

This thesis consists of five chapters. Chapter 1 provides a brief introduction to multizone cooling system with chilled-water storage. The principles of dynamic programming, conventional dynamic programming procedure, and state increment dynamic programming procedure are explained in detail in Chapter 2. Efficient operation of a bilinear two-zone cooling system is studied and the result for a system with two environmental zones is presented in Chapter 3. In chapter 4, first the aggregation/disaggregation sequential approach is introduced, the application to eight zone cooling system is shown, then a block by block method is developed for multizone cooling system with larger size and the result of this technique as applied to four zones is discussed. Conclusions are drawn in Chapter 5.

Chapter 2

The Principles of Dynamic Programming

2.1 Introduction

Optimal control is one of the most active research areas of modern technology, but it has been applied in cooling systems only in recent decades, after numerous techniques for solving this type of problem have been developed. Dynamic programming[7], originally developed by Richard Bellman, has long been recognized as an extremely powerful approach to solving optimization problems. The basic approach of dynamic programming is ideally suited for implementation on a digital computer. The standard computational algorithm based on dynamic programming is very desirable from a number of points of view, including the generality of problems to which it can be applied, the nature of the solution that is obtained, and the ease with which it can be programmed.

Despite the attractive features of the standard algorithm, its applicability thus far has been limited to relatively simple cases. This is due to the large computational requirements of this algorithm. The most severe restriction is generally the amount of high-speed memory required to implement the basic calculations. Another difficulty is the amount of computing time required to obtain the complete solution. Thus, while dynamic programming is frequently used as an analytical and conceptual tool, the computational difficulties associated with the standard algorithm have severely limited its application to large-scale optimization problems.

As a result of this situation, a number of researchers, including Bellman and his colleagues, have been motivated to develop new computational procedures which retain the desirable properties of the standard algorithm but reduce computational requirement [8]. State increment dynamic programming [9] is one of the procedures that greatly reduces the high-speed memory and computing time requirement. State-increment dynamic programming has been a powerful yet practical method applied in many optimal control fields.

2.2 Dynamic Programming Fundamentals

2.2.1 Problem Formulation

The optimization problems to which dynamic programming applies are variously called dynamic optimization problem. The essential elements of the problem are system equations, which describe the process being controlled; the performance criterion, which evaluates a particular control policy; and the constraints, which place restrictions on the system operation.

The system equations are a set of relations between three types of variables. The first type is the state variable, these variables provide a complete description of the dynamic behavior of the system. The second type is the control variable. These variables are the decisions that are to be made in an optimum fashion. The third type is the stage variable, which determines the order in which controls are applied, it is generally taken to be time. The equations are taken to be a set of nonlinear time-varying differential equation:

$$\dot{x} = f(x, u, t) \tag{2.1}$$

where x = n-dimensional state vector

u = q-dimensional control vector

t = stage variable, assumed to be time

In order to represent these equations on a digital computer, some finite integration formula must be used to approximate the differential equation. The simplest approximation is

$$x(t + \delta t) = x(t) + f[x(t), u(t), t]\delta t \tag{2.2}$$

where δt = time increment over which control u(t) is applied.

The performance criterion, which determines the effectiveness of a given control function, is taken to be a cost function that is to be minimized. This cost function has the general; variational form, which is

$$J = \int_{t_0}^{t_f} l[x(\sigma), u(\sigma), \sigma]\delta\sigma + \psi[x(t_f), t_f] \tag{2.3}$$

where t_0 = initial time

t_f = final time

σ = dummy variable for time

l = loss function; cost function per unit time

ψ = scalar functional for final-value cost

Again, if this equation is to be represented on a digital computer, the integration must be approximated by finite formula. The simplest formula is

$$\int_{t}^{t+\delta t} l[x(\sigma), u(\sigma), \sigma]d\sigma \simeq l[x(t), u(t), t]\delta t \tag{2.4}$$

The constraints in the problem are expressed as

$$x \in X(t) \tag{2.5}$$

$$u \in U(t) \tag{2.6}$$

where X, the set of admissible states, can vary with time t, and where U, the set of admissible controls can vary with x and t.

The optimization problem can then be stated as follows:

Given:

1. A system described by Eq.(2.1)
2. constraints that $x \in X(t)$, $u \in U(t)$
3. An initial state x(0)

Find:

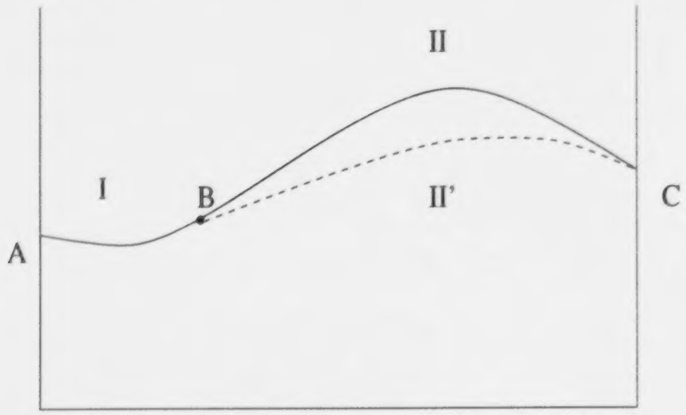


Figure 2.1: Illustration of the principle of optimality.

The control function u(t), $t_0 \leq t \leq t_f$, such that the performance criterion in Eq.(2.3) is minimized and all the constraints are satisfied.

2.2.2 Bellman's Principle Of Optimality

The heart of dynamic programming is Bellman's principle of optimality. This principle makes it possible to construct a computational procedure that retains the desirable properties of the enumeration procedure. The principle of optimality can be stated as follows: given an optimal trajectory from point A to point C, the portion of the trajectory from any intermediate point B to point C must be the optimal trajectory from B to C.

In Figure 2.1, if the path I-II is the optimal trajectory from A to C passing through B at time t_1, $t_0 \leq t_1 \leq t_f$, then according the principle of optimality, path II is the optimal path from B to C over time interval(t_1, t_f). The proof by contradiction for this case is immediate: assume that some other path, such as II', is the optimal path from B to C. Then, path I-II' has less cost than Path I-II. However, this contradicts the fact that I-II is the optimal path from A to C, and hence II must be the optimal path from B to C.

2.2.3 Derivation Of The Iterative Functional Equation

In order to write the iterative equation, the minimum cost function I(x,t), must first be defined. This function determines the minimum cost that is incurred in going to the final time if the present time is t and if the present state is x. It is defined for all admissible states $x \in X$ and for all time $t_0 \leq t \leq t_f$. The defining equation is

$$I(x,t) = \min_{\substack{u(\sigma)\in U \\ t\leq\sigma\leq t_f}} \int_t^{t_f} l[x(\sigma), u(\sigma), \sigma]d\sigma + \psi[x(t_f), t_f], \tag{2.7}$$

where $x(t) = x$.

The iterative equation can be deduced from the principle of optimality. If the approximations of Equations(2.2) and (2.4) are used, this equation is obtained as

$$I(x,t) = \min_{u\in U} l(x,u,t)\delta t + I[x + f(x,u,t)\delta t, t + \delta t]. \tag{2.8}$$

The interpretation of this equation is that the minimum cost at a given state x and the present time t is found by minimizing, through the choice of the present control u, the sum of $l[x,u,t]\delta t$, the cost over the next time interval δt, plus $I[x + f(x,u,t)\delta t, t + \delta t]$, the minimum cost of going to t_f from the resulting next state, $x + f[x,u,t]\delta t$.

This iterative equation is solved backwards in time because $I(x,t)$ depends on values of the minimum cost function at future times. Consequently the iterations begin by specification of the minimum cost function at the final time, t_f. Using Eq.(2.7),

$$I(x,t_f) = \psi(x,t_f) \tag{2.9}$$

The minimum cost function for all x at t can be evaluated by iteratively solving Eq.(2.8) with Eq.(2.9) as a boundary condition. The optimal control at every x and t, denoted as $\hat{u}(x,t)$, is obtained at the value of u(t) which minimizes Eq.(2.8) for the given x and t.

2.3 Conventional Dynamic Programming Computational Procedure

2.3.1 Computational Procedure

During conventional dynamic programming computational procedure, the stage variable can either be continuous or discrete. If it is continuous, it is denoted as t, defined over interval $t_0 \leq t \leq t_f$; while if it is discrete, it is defined as the sequence $k = 0, 1, 2, ..., K$. In order to implement the procedure of this section, when the stage variable is continuous, it is quantized into increments, denoted as $\triangle t$. The quantized values of t that lie in the range over which t is defined, $t_0 \leq t \leq t_f$, can then be indexed by the discrete sequence $k = 0, 1, ..., K$, where the value of t corresponding to k is given by

$$t = t_0 + k\triangle t \tag{2.10}$$

and where

$$K\triangle t = t_f - t_0 \tag{2.11}$$

The optimization problem can then be stated as follows:

1. A system described by $x(k+1) = g[x(k), u(k), k]$, where g is an n-dimensional vector functional
2. Constraints that $x \in X(k)$, $u \in U(x,k)$
3. An initial state x(0)

Find:

The control sequence u(0),u(1),...u(K) that minimize

$$J = \sum_{k=0}^{K} l[x(k), u(k), k]$$

while satisfying the constraints.

The iterative relation, Eq.(2.7) can be rewritten as

$$I(x,k) = \min_{u \in U} l[x,u,k] + I[g(x,u,k), k+1] \tag{2.12}$$

The boundary condition is then

$$I(x,K) = \psi(x,t_f) \tag{2.13}$$

The range of the state variables can be restricted by relation such as

$$\beta_i^- \le x_i \le \beta_i^+ \quad (i = 1,2,...,n) \tag{2.14}$$

In order to apply the dynamic programming computational procedure, there must be a finite number of admissible states. This requirement is usually met by quantizing these variables. Within the range determined by Eq.(2.14) each state variables x_i is quantized with uniform increment $\triangle x_i$, these increments could be nonuniform, but needless notational complications would arise. The quantized values of x_i are thus

$$x_i = \beta_i^- + j\triangle x_i \tag{2.15}$$

where $\quad j = 0, 1, ..., N_i$

$$N_i \triangle x_i = \beta_i^+ - \beta_i^-$$

The set of all X, where each component x_i is quantized according to Eq.(2.15) will from now on be referred to as X, the set of quantized admissible states.

The control variables can also be restricted as

$$\alpha_j^- \le u_j \le \alpha_j^+ \quad (j = 1, 2, ..., q). \tag{2.16}$$

The control variables can also be quantized according to relations similar to Eq.(2.15). However, in this section it is sufficient to assume that there are a finite number of admissible controls. The set of admissible controls U, can thus be denoted as

$$U = u^{(1)}, u^{(2)}, ..., u^{(M)} \tag{2.17}$$

where M is the total number of admissible controls.

Once all these have been determined, it is possible to compute optimal control by iterative application of the functional equation, Eq.(2.12). Consider a quantized state $x \in X$ at stage $(K-1)$. At this state each of the admissible control $u^{(m)} \in U$ is applied. For each of these controls the cost of control over the next stage can be determined as

$$L_1^{(m)} = l[x, u^{(m)}, K-1] \quad (m = 1, 2, ..., M) \tag{2.18}$$

Next, for each of these controls the next state at stage K is determined from the system equation,

$$x^{(m)}(K) = g[x, u^{(m)}, K-1] \quad (m = 1, 2, ..., M) \tag{2.19}$$

The next step is to compute the minimum cost at stage K for each of the state $x^{(m)}$. However, in general a particular state $x^{(m)}$ will not lie on one of the quantized states $x \in X$ at which the optimal cost $I(x, K)$ is defined. In fact, it may lie outside of the range of admissible states determined by Eq.(2.14). In the latter case the control is rejected as a candidate for the optimal control for this state and stage.

If a next state $x^{(m)}$ does fall within the range of allowable states, but not on a quantized value, then it is necessary to use some type of interpolation procedure to compute the minimum cost function at these points. In general, the interpolation procedure consists of using a low-order polynomial in the n state variables to approximate the minimum cost function at quantized states, $x \in X$. The determination of the coefficients is made according to some criterion, such as least-squares fit. Computation procedures for calculating the coefficients are relatively simple and well-known.

Assume, then, that the values of the minimum cost at the state $x^{(m)}$ can be expressed as a function of the values of the optimal cost at quantized state $x \in X$.

$$I[x^{(m)}, K] = P[x^{(m)}, K, I(x, K)] \quad (x \in X) \tag{2.20}$$

where P is minimum cost at the next state, which is found by interpolation using the values of I(x,k) at quantized states.

The total cost of applying control $u^{(m)}$ at state x, stage $(K-1)$, can then be written as

$$J_1^{(m)} = l[x, u^{(m)}, K-1] + I[x^{(m)}, K] \tag{2.21}$$

The minimization can be achieved by simply comparing the M quantities. According to the functional equation, the minimum value will be the minimum cost at state x, stage $(K-1)$.

$$I[x, K-1] = \min_{u^{(m)} \in U} l[x, u^{(m)}, K-1] + I[x^{(m)}, K] \tag{2.22}$$

The optimal control at this state and stage, $\hat{u}[x, K-1]$, is the control $u^{(m)}$ for which the minimum in Eq.(2.22) is actually taken on.

This procedure is repeated at each quantized $x \in X$ at stage $K-1$. When this has been done, $I(x, K-1)$ and $\hat{u}(x, K-1)$ are known for all $x \in X$. It is now possible to compute $I(x, K-2)$ and $\hat{u}(x, K-2)$ for all $x \in X$ based on knowledge of $I(x, K-1)$.

The general iterative procedure continues this process. Suppose that $I(x, k+1)$ is known for all $x \in X$. Then $I(x, k)$ and $\hat{u}(x, k)$ are computed for all $x \in X$ from

$$I(x, k) = \min_{u^{(m)} \in U} l[x, u^{(m)}, k] + I[x^{(m)}, k+1] \tag{2.23}$$

where $x^{(m)}$ is determined from

$$x^{(m)} = g[x, u^{(m)}, k] \tag{2.24}$$

and where $I[x^{(m)}, k+1]$ is computed by interpolation on the known values $I[x, k+1]$ for $x \in X$

$$I[x^{(m)}, k+1] = P[x^{(m)}, k+1, I(x, k+1)] \tag{2.25}$$

The optimal control $\hat{u}(x, k)$ is the control for which Eq.(2.23) takes on the minimum. The iterative procedure begins by computing $\hat{u}(x, K-1)$ and $I(x, K-1)$ from the given boundary conditions, $I(x, K)$ and it continues until $\hat{u}(x, 0)$ and $I(x, 0)$ have been computed.

2.3.2 Computational Requirements

Before using dynamic programming to solve an actual problem on a digital computer, it is necessary to determine how much computational effort is required to solve the problem. The requirement are given in terms of the amount of memory needed to store both program and data and the amount of time required to perform the computations. If for a given computer, either there is not sufficient memory available or if the cost of computing time exceeds the economic value of the results, then the problem could not be run on that computer. If this is true for all available computers, then the problem, while solvable in principle , must be regarded as unsolvable in a practical sense.

The most commonly encountered barrier to the use of dynamic programming is the high-speed memory requirement. This requirement refers to the number of locations in the high-speed access memory which must be available during the computations to store intermediate result, in addition to the locations needed for the program, the compiler and other special functions. The dynamic programming procedure requires that sufficient data be stored to specify $I(x,k)$ for all $x \in X$ at a single value of k. In general, this is done by storing one value of $I(x,k)$. Since data type double is used for this value (one double is 8 bytes), the high speed memory required is then

$$N_h = \prod_{i=1}^{n} N_i * 8 \; bytes \tag{2.26}$$

where N_i is the number of quantized values of ith variable and n is the total number of state variables.

A second storage problem arises in retaining the results of the computation. there are K stages, then the number of values of $\hat{u}(x,k)$ and of $I(x,k)$ which are computed is N_c where

$$N_c = \prod_{i=1}^{n} N_i * K * 8 \; bytes \tag{2.27}$$

Although this number can be extremely large, currently available low-speed memory devices are capable of storing this much information reliably and at a reasonable cost.

The computing time requirement is also related to the vast number of results that are obtained. If N_c values of $\hat{u}(x,k)$ and $I(x,k)$ are obtained and if it takes $\triangle t_c$ seconds for each computation, then the total amount of computing time is T_c

$$T_c = \prod_{i=1}^{n} N_i * K * \triangle t_c \; secs \tag{2.28}$$

For example, if there are 3 state variable ($n = 3$), and if there are 100 quantization levels in each variable($N_i = 100, \; i = 1, 2, 3$), 50 stages and the time for each computation is $50 * 10^{-6}$ secs then the high-speed memory approximately:

$$N_h = (100)^3 * 8 = 8 * 10^6 bytes \tag{2.29}$$

The low-speed memory requirement is

$$N_c = (100)^3 * 50 * 8 = 4 * 10^8 bytes \tag{2.30}$$

The computing time requirement is

$$T_c = (100)^3 * 50 * 50 * 10^{-6} = 2500 secs = 41.67 minutes \tag{2.31}$$

Although N_c and T_c are reasonable number, N_h is quite large for high speed memory.

Despite the great advantage over enumeration, the preceding results of the conventional dynamic programming computational procedure indicate that computational requirement can be extremely large for even moderately sized problems and in most case, the high-speed memory requirement becomes excessive before the other requirements do.

2.4 State Increment Dynamic Programming

2.4.1 Basic Concepts of State Increment Dynamic Programming

State increment dynamic programming[15],[16],[17],[18] is an optimization procedure that has all the desirable properties of the conventional dynamic programming computational procedure and yet has much smaller computational requirements. In particular, the high-speed memory requirement is always reduced significantly. Similar savings in computing time and low-speed memory can also be achieved in certain cases.

The optimization problem to which the state increment dynamic programming computational procedure is applied is most conveniently formulated in the continuous-time case. The stage variable t, thus varies continuously over the interval $t_0 \leq t \leq t_f$. The problem is stated as:

1. A system equation as $\dot{x} = f(x, u, t)$
2. Constraints on state and control variables as $x \in X(t)$, $u \in U(x, t)$
3. An initial state, $x(t_0)$

Find:

A control function u(t), $t_0 \leq t \leq t_f$, such that the performance in Eq.(2.3) is minimized and all the constraints are satisfied. The iterative functional equation becomes

$$I(x,t) = \min_{u \in U}(l[x,u,t]\delta t + I[x + f(x,u,t)\delta t, t + \delta t]) \tag{2.32}$$

and the boundary condition is

$$I(x, t_f) = \psi(x, t_f) \tag{2.33}$$

The state variables and control variables are restricted and quantized in the similar manner of conventional dynamic programming. The state variable t is also quantized with an increment size $\triangle t$.

The fundamental difference between state increment dynamic programming and conventional dynamic programming is in the method for determining δt, the time interval over which a given control is applied. In conventional dynamic programming the total time interval over which optimization is performed. $t_0 \leq t \leq t_f$, is quantized into uniform increment $\triangle t$, and optimal control is computed only at these quantized values of t. In computation of optimal control according to the iterative functional equation, the next state is always taken to occur $\triangle t$ seconds after the present state. Consequently, every admissible control is considered to be applied for $\triangle t$ seconds. Therefore, in conventional dynamic programming, $\delta t = \triangle t$ i.e. the time over which a given control is applied is fixed at $\triangle t$ time interval between successive computations of optimal control. In state increment dynamic programming, on the other hand, the determination of these two time intervals is made independently. The time interval between successive computations of optimal control may or may not be fixed. However, δt varies with each control which is applied. The interval δt is determined as the minimum time interval required for any one of the n state variables to change by one increment. This is the source of the name “state increment dynamic programming”. If $\triangle x_i$ is the increment in the ith state variable and if control u is being applied at state x and time t, then δt can be expressed as

$$\delta t = \min_{i=1,2,\ldots,n} \frac{\triangle x_i}{|f_i(x,u,t)|} \tag{2.34}$$

As a result, the next state, $x + f(x,u,t)\delta t$, is always close to the present state. Specifically, it lies on the surface of an n-dimensional hypercube centered at x, with length $2\triangle x_i$ along the x_i-axis. In the general case of conventional dynamic programming, on the other hand, because control is always applied for the fixed increment$\triangle t$, the next state can occur anywhere in the space of admissible states. This difference enables state increment dynamic programming to reduce the computational requirements from those of conventional dynamic programming. If the next state is close to the present state, as in state increment dynamic programming, in order to perform the interpolation of the minimum cost function, as required in the iterative functional Eq.(2.7), it is necessary to store values of the minimum cost function at only those quantized state near the present state. However, if the next state can occur anywhere in the space of admissible states, as in conventional dynamic programming, then, it is necessary to store values of the minimum cost function for every quantized state $x \in X$.

A significant overall saving in high-speed memory requirement can also be achieved by only processing the data in units called blocks. Blocks are defined by partitioning the $(n+1)$-dimensional space containing the n state variable and time into rectangular sub-units. Each block covers w_i increment along the x_i-axis and $\triangle T$ seconds along the t-axis. A particular block is denoted by the largest values of the coordinates that are contained within the block. The block $B(j_1, j_2, j_3, ..., j_n)$ contains values of x and t such that

$$(j_i - 1)w_i\triangle x_i \leq x_i - \beta_i^- \leq j_i w_i \triangle x_i \quad (i = 1, 2, ..., n) \tag{2.35}$$

$$(j_0 - 1)\triangle T \le t - t_0 \le j_0 \triangle T \qquad (2.36)$$

where $j_i = 1, 2, ..., J_i$

$$J_i w_i \triangle x_i = \beta_i^+ - \beta_i^-$$

$$i = 1, 2, ..., n$$

$$j_0 = 1, 2, ..., J_0$$

$$J_0 \triangle T = t_f - t_0$$

As indicated by the above equations, the boundaries between blocks are considered to be in both blocks. The number w_i are taken to be small integers, usually between 2 and 5. The value of $\triangle T$, on the other hand, is taken to be considerably larger than the average value of δt as determined by Eq.(2.34). For a one-dimensional example the blocks are two-dimensional rectangles. A typical set of blocks is illustrated in Figure 2.2 where $w_1 = 3$, $\beta_1^+ = 12\triangle x_1$, $\beta_1^- = 0$, $t_f = 4\triangle T$ and $t_0 = 0$. Note that each block contains $w_1 + 1 = 4$ quantized values of x_1.

In state increment dynamic programming, computations are done for one block at a time on the assumption that optimal trajectories never leave the block. Since each block contains only $(w_i + 1)$ increments along state x_i, there are a relatively small number of quantized states contained within a single block. Furthermore, the method of choosing δt implies that, unless the present state is on the boundary of the block, the quantized states needed for interpolation of the minimum cost function at the next state are within the block. Therefore, the number of high-speed memory storage locations required for computation of optimal control in one block is quite small. Also since $\triangle T$ is relatively large, a large number of computations of optimal control take place within the block

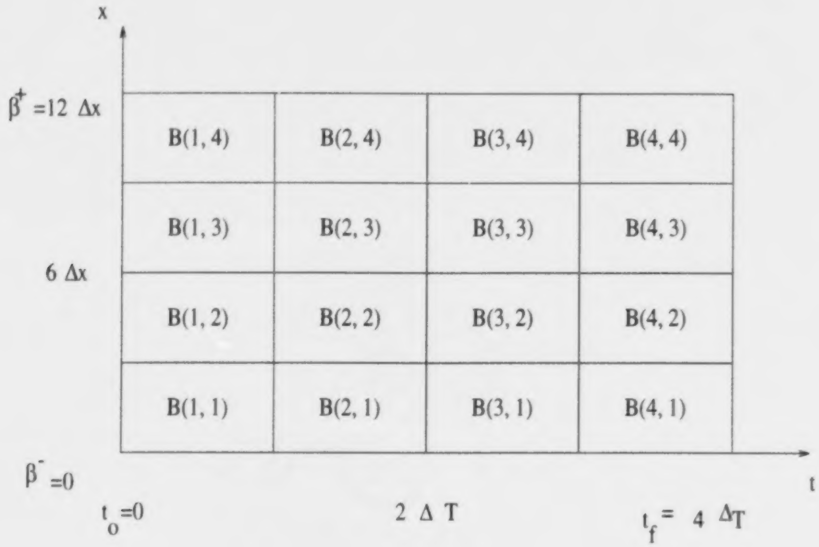

x
β+ =12 Δx
6 Δx
β- =0
B(1, 4)
B(2, 4)
B(3, 4)
B(4, 4)
B(1, 3)
B(2, 3)
B(3, 3)
B(4, 3)
B(1, 2)
B(2, 2)
B(3, 2)
B(4, 2)
B(1, 1)
B(2, 1)
B(3, 1)
B(4, 1)
t
t0 =0
2 Δ T
tf = 4 ΔT

Figure 2.2: Blocks for one-dimensional example.

using a small number of high-speed memory storage locations.

2.4.2 Computation Procedure Within a Block in the General Case

In the general case, the time interval $\triangle T$ is further divided into small increments $\triangle t$, a set of quantized times is determined

$$t = t_0 + (j_0 - 1)\triangle T + s\triangle t \tag{2.37}$$

where

$s = 0, 1, 2, ..., S$

$S\triangle t = \triangle T$

Typical values of S range from 5 to 15. Optimal control is computed at each quantized $x \in X$ for each quantized t of the form above. The operations at a given x and t take place as follows:

Each admissible control $u^{(m)} \in U$ is applied. For each control, the time over which the control is applied is determined as Eq.(2.34), the corresponding next state is computed as

$$x^{(m)} = x + f(x, u^{(m)}, t)\delta t^{(m)} \tag{2.38}$$

Interpolation formulas such as (2.25), but restricted to the current block of quantized states, are used to obtain

$$I(x + f(x, u^{(m)}, t)\delta t^{(m)}, t + \delta t^{(m)}] \tag{2.39}$$

Then the iterative functional equation becomes

$$I(x, t) = \min_{u \in U} l(x, u^{(m)}, t)\delta t + I[x + f(x, u^{(m)}, t)\delta t, t + \delta t] \tag{2.40}$$

The optimal control at state x, time t, denoted as $\hat{u}(x, t)$ can be evaluated directly as that control for which I(x,t) takes on its minimum value.

2.4.3 Procedure for Processing Blocks

In the procedure describe so far, it has been assumed that the next state is always in the same block as the present state. In order to allow optimal trajectories to pass from block to block, the computations near the boundaries of the block must be slightly modified. For two adjacent blocks, one of which has been processed and the other has not, a sequence of optimal points has been computed along the boundary between these blocks. In the computation of the latter block, the optimal points are stored in the high-speed memory, and no new optimal points are computed along the boundary. Values of the optimal cost function for these points are used in evaluating the cost of those controls applied at states within one increment of the boundary for which the next state lies on this boundary. It is then true that an optimal trajectory can pass from a state in the interior of the block being computed to the boundary with a previously computed block. From this boundary the optimal trajectory can then travel further into the previously computed block. In this manner, transitions from a given block to an adjacent block that has been previously processed can take place without constraint, provided that optimal points on the boundary are available.

In some problems of physical significance, it is possible to obtain enough insight about the system so that it is possible to determine the direction which optimal trajectories are most likely to take. This direction is called the "preferred" direction of motion. When a preferred direction of motion can be specified, adjacent blocks in a given time interval are processed in order such that the preferred direction is from the block processed later to the one processed first. In this manner, interblock transitions in the preferred direction take place without constraint.

Since the preferred direction of motion cannot always be determined exactly before the computations begin, it is necessary to have available techniques for allowing interblock transitions into a block that has not been previously computed. These techniques can be either very complex, if the preferred direction is not well-defined, or else quite simple, if it is known accurately. Since this case is not used during this research, further discussion (see chapter 6 of *State Increment Dynamic Programming*[9]) is not provided here.

2.4.4 Computational Requirement

Since in the state increment dynamic programming procedure the next state is always in the same block as the present state, the high-speed memory requirement is considerably less than that of the conventional procedure. In the general case, if a block covers $w_i \triangle x_i$ units along the ith axis, so that there are $(w_i + 1)$ quantization levels in each block, then the high-speed memory requirement is

$$N_h = \prod_{i=1}^{n} (w_i + 1) * 8 \; bytes \tag{2.41}$$

For a system with three state variable and 100 quantization levels in each state variable, block size is $(w_i + 1) = 5$, $S + 1 = 11$, $i = 1, 2, 3$ the high speed memory requirement is

$$N_h = 5 * 5 * 5 * 8 = 125 * 8; bytes$$

which is far less than $10^6 * 8$ bytes corresponding to requirement for conventional dynamic requirement. For the low-speed memory, a total of $N_{c/b}$ computations per block is

$$N_{c/b} = S \prod_{i=1}^{n} w_i * 8 \; bytes \tag{2.42}$$

If the number of blocks processed is N_b, then the low-speed memory requirement is given by

$$N_c = N_b S \prod_{i=1}^{n} w_i * 8 \; bytes \tag{2.43}$$

if $\triangle t_c$ seconds are required for each computation, the total computing time is

$$T_c = N_b S \triangle t_c \prod_{i=1}^{n} w_i \; secs \tag{2.44}$$

Chapter 3

Efficient Operation of a Bilinear Two-zone Cooling System with Chilled-Water Storage

3.1 Introduction

The cooling system under study has two environmental zones. Figure 3.1 shows each zone has its own fan-coil unit which consists of a cooling coil, a circulating fan and supply and return pipes. Pipes carry water to the coil and back to the storage tank. A compression-cycle refrigeration unit (chiller) removes heat from the storage tank.

Of particular interest is the case where the storage capacity is substantial, for then the chiller can be mainly operated at times when the zone cooling loads are low, e.g. to take advantage of off-peak electrical energy that may be offered at a discount price by the utility company. Under the assumption that the zone cooling-load profiles are going to be periodic, with 24-hour period, the principle

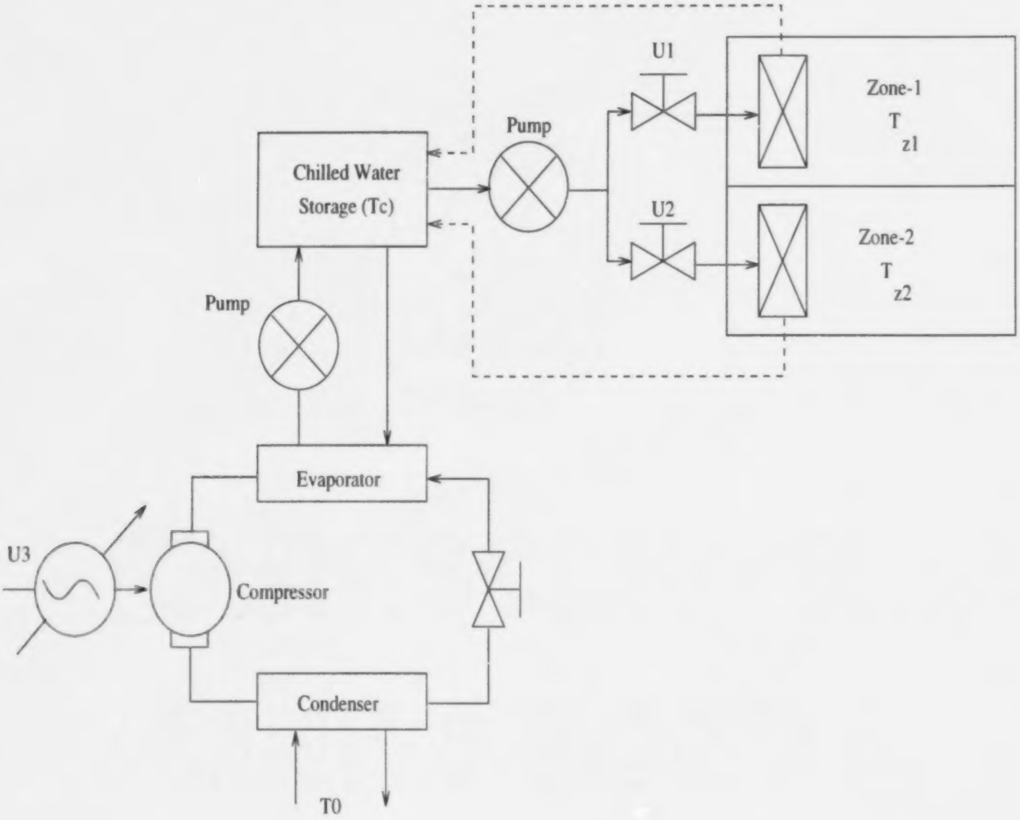

U1
Zone-1
T
z1
Pump
Chilled Water
Storage (Tc)
U2
Zone-2
T
z2
Pump
Evaporator
U3
Compressor
Condenser
T0

Figure 3.1: Schematic diagram of a two-zone cooling system.

goal of the research is development of controller algorithms that can be expected to maintain the zone temperatures within comfort limits, despite diurnal and stochastic fluctuations in ambient conditions and cooling loads, at the same time controlling the chiller in such a way as to minimize the cost of purchased energy. This solution is not obvious, since lower temperature chilled water allows for lower mass flow rates, but implies low coefficient of performance (COP) of the chiller and hence higher energy consumption for the chiller. On the other hand, higher temperature chilled water can still allow for zone cooling loads to be met, but only at the cost of high mass flow rates through the coils with high pumping power and, indirectly, high capital cost.

In [1] it was shown that an optimization method, state increment dynamic programming, simplified through the use of state feedback linearization and decoupling, and augmented with a label-tracing procedure to identify the periodic but unknown initial/terminal state, provides a reliable way of solving this problem. After the optimal periodic temperature trajectories for the storage tank and zones have been found, simple feedback controllers can then use them as reference trajectories, tracked by the actual temperatures.

3.2 Problem Formulation

For the particular problem shown in Figure 3.1, the stage variable is time t. The state variables are T_i, (i=1,2) the temperatures of each zone, and T_c, temperature of storage tank. There are three control variables, u_1, mass flow rate of zone1, u_2, mass flow rate of zone2, and u_3, input power of the chiller. The equations of state for the system can be written as heat balance equations for the three lumped capacities, as shown below:

$$C_{z1}\dot{T}_{z1} = -u_1\zeta(T_{z1} - T_c) + a_{z1}(T_e - T_{z1}) + q_{z1}(t) \tag{3.1}$$

$$C_{z2}\dot{T}_{z2} = -u_2\zeta(T_{z2} - T_c) + a_{z2}(T_e - T_{z2}) + q_{z2}(t) \tag{3.2}$$

$$C_c\dot{T}_c = -u_3COP + u_1\zeta(T_{z1} - T_c) + u_2\zeta(T_{z2} - T_c) + a_c(T_e - T_c) \tag{3.3}$$

a_{z1}: zone1 heat loss coefficient

a_{z2}: zone2 heat loss coefficient

a_c: heat loss coefficient of storage tank

C_{z1}: heat capacity of zone1

C_{z2}: heat capacity of zone2

C_c: heat capacity of storage tank

q_{z1}: cooling load of zone1

q_{z2}: cooling load of zone2

T_e: temperature of environment

ζ: heat exchanger coefficient

COP: coefficient of performance of the chiller

The first terms on right side of Eq.(3.1) and (3.2) are heat transfer rates from zone i to circulating chilled water with pumping rate u_i, the second terms are heat transfer rates from the environment to each zone, and the third terms are cooling loads generated by occupants and equipment within each zone. The third equation represents the heat balance for the chilled-water storage tank, where the first term on the right side is rate of heat removal by the chilling unit, with coefficient of performance for refrigeration modeled as

$$COP = \begin{cases} (COP_{max} - 1)[1 - (T_0 - T_c)/T_{max}], & \text{if } T_0 - T_c < T_{max} \\ 0, & \text{if } T_0 - T_c > T_{max} \end{cases} \tag{3.4}$$

COP_{max} is the maximum achievable heat pump coefficient of performance, T_0 is the temperature of the water or air used for cooling the condenser, and T_{max} is the maximum temperature difference achievable between condenser and evaporator. The notable fact is that the state equations are bilinear, linear in the state variables and in the control variables, but not jointly linear in both kinds of variables. If new state variables are defined by scaling the temperatures as

$$x_1 = \sqrt{C_{z1}}\,T_{z1}; \qquad x_2 = \sqrt{C_{z2}}\,T_{z2}; \qquad x_3 = \sqrt{C_c}\,T_c$$

the corresponding state dynamic equations can be written as

$$\dot{x} = Ax + u_1 B_1 x + u_2 B_2 x + u_3 B_3 x + Cu + d(t) \tag{3.5}$$

where A is the diagonal matrix

$$A = diag(-a_{z1}/C_{z1}, -a_{z2}/C_{z2}, -a_c/C_c) \tag{3.6}$$

the B matrices are the symmetric matrices

$$B_1 = b_1 \begin{bmatrix} 1 & 0 & -\sqrt{r_1} \\ 0 & 0 & 0 \\ -\sqrt{r_1} & 0 & r_1 \end{bmatrix}; B_2 = b_2 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -\sqrt{r_2} \\ 0 & -\sqrt{r_2} & r_2 \end{bmatrix}; B_3 = b_3 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$

where $b_1 = -\zeta/C_{z1}$, $b_2 = -\zeta/C_{z2}$, $b_3 = -(COP_{max} - 1)/C_c T_{max}$, $r_i = C_{zi}/C_c$

The control vector u is

$$u = [u_1, u_2, u_3]^t$$

and C and d(t) are given by

$$C = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & c_{33} \end{bmatrix}; \qquad d(t) = \begin{bmatrix} (a_{z1}T_e + q_{z1}(t))/\sqrt{C_{z1}} \\ (a_{z2}T_e + q_{z2}(t))/\sqrt{C_{z2}} \\ a_c T_e(t)/\sqrt{C_c} \end{bmatrix} \qquad (3.8)$$

where

$$c_{33} = -(COP_{max} - 1)(1 - T_0/T_{max})/\sqrt{C_c}$$

It is clear that system Eq.(3.5) will be stable with any fixed choice of control vector, since the system matrix

$$A + u_1 B_1 + u_2 B_2 + u_3 B_3$$

is symmetric with real eigenvalues, orthogonal real eigenvectors, and adding Eq.(3.1) through (3.3) yields that the rate of change of system stored energy cannot grow without bound.

$$\frac{d}{dt}[C_{z1}T_{z1} + C_{z2}T_{z2} + C_cT_c] = \sum_{i=1}^{2}[a_{zi}(T_e - T_{zi}) + q_{zi}] + a_c(T_e - T_c) - u_3COP$$

The optimal control problem to be solved is minimization of the performance criterion

$$J = \int_0^{24} [p(t)(R_1u_1^2 + R_2u_2^2 + u_3) + a_1(x_1 - x_{1s})^2/C_{z1} + a_2(x_2 - x_{2s})^2/C_{z2}]dt \quad (3.9)$$

where p(t) is the energy price factor which can be changed periodically. The first and second terms in the integrand are the cost of electric power consumption by circulation pumps (assuming incompressible laminar flow), the third term is the cost of electric power input to the compressor drive motor, and the last two terms are weighted index of discomfort from zone actual temperature deviations away from setpoint or desired temperature.

One day is considered as a period, related temperature of outside environment (T_e) and cooling loads, approximated as hourly constant, are forecasted at time t=0. Typical daily cooling loads acting on the zones are shown in Figure 3.2. The price of purchased energy was allowed to vary periodically as well. Constraints are specified on temperature of both zones and storage tank as: $23^oC < T_{zi} < 31^oC$ $i = 1, 2$; $8^oC < T_c < 15^oC$ for all t, value assumed for $T_o = 15^oC$, initial state x(0) unknown, but subject to periodic boundary conditions

$$x(0) = x(24) \tag{3.10}$$

The constraints of control variables are $u_i(t) \geq 0$ for all t, $i = 1, 2, 3$. No maximum constraints on control inputs are assumed here, since very large values are penalized due to their presence in Eq.(3.9). Despite the fact that u_3 appears only linearly, optimal solutions will always be bounded because excessive u_3 values cause the coefficient of performance COP to decrease rapidly, effectively penalizing u_3 at a faster-than-linear rate. In fact, one can draw certain intuitive conclusions about what an optimal strategy should accomplish. It should

1. Avoid excessive exchanger sizes and pumping rates, by keeping $(T_{zi} - T_c)$ high.
2. Use the chilling unit with COP values as high as possible, by keeping $(T_0 -$

T_c) low.

Obviously these goals are conflicting, minimization of Eq.(3.9) with suitable weights provides for optimal compromise.

3.3 Linearization by State Feedback

In order to find control variables conveniently, state feedback linearization is used. Treating the three state variables as observable outputs, the method of exact linearization by state feedback [11] can be easily applied. The result is obtained by writing the equation of state in the form

$$\dot{x} = Ax + E(x)u + d(t) \tag{3.11}$$

where

$$E(x) = [B_1 x + c_1 \,|B_2 x + c_2|\, B_3 x + c_3] \tag{3.12}$$

and where $C = [c_1|c_2|c_3]$.

This matrix can be simplified for the class of problems considered here, by noting that the B matrices given in Eq.(3.7) are symmetric and of rank one, hence can be written as

$$B_i = b_i[w_i][w_i]^t \tag{3.13}$$

where the vectors w_i are

$$w_1 = \begin{bmatrix} 1 \\ 0 \\ -\sqrt{r_1} \end{bmatrix}; \quad w_2 = \begin{bmatrix} 0 \\ 1 \\ -\sqrt{r_2} \end{bmatrix}; \quad w_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.14}$$

Therefore

$$E(x) = [b_1 w_1 g_1(x)\,|b_2 w_2 g_2(x)|\,c_3 + b_3 w_3 g_3(x)] \tag{3.15}$$

with $g_i(x) = w_i^t x$, and for any state x of interest E(x) is nonsingular, i.e.

$$det E(x) = b_1 b_2 g_1(x) g_2(x)[c_{33} + b_3 g_3(x)] \neq 0 \tag{3.16}$$

since if E(x) is singular g_1 or g_2 or $c_{33} + b_3 g_3$ should be zero. This implies that $T_{z_1} = T_c$ or $COP = 0$, which will never occur during typical operations for the system under study.

Therefore let

$$u = -E^{-1}(x)[Ax - v], \tag{3.17}$$

which implies that the feedback system with new input vector v satisfies the linear, decoupled state equation

$$\dot{x} = v + d(t) \tag{3.18}$$

This representation simplifies the dynamic programming solution of the optimization problem. With the method of state increment dynamic programming used here, to find an optimal trajectory segment from a node (quantized state), all transitions to reachable nearest-hypercube nodes are examined and their associated costs compared to find the minimum. This can easily be done if the system is transformed to the form of Eq.(3.18). For then a set of v vectors can be easily found that achieve each of the transitions being considered, under the condition that $d(t)$ is given as a forecast. The corresponding u vectors can be found according to Eq.(3.17). Substituting u vectors in Eq.(3.9) the costs of these transition can be obtained and compared.

3.3.1 Optimization by State Increment Dynamic Programming

At the operating point where during occupied hours the setpoint of zone's temperature is 24^oC and during unoccupied time is 30^oC, the maximum boundary of zone temperature is specified as 31^oC and the minimum boundary of zone temperature is 23^oC. The temperature range of the storage is constrained between 8^oC to 15^oC. As a result, the constrained volume of state space is just the volume bounded by these limits. Quantizing state variables, there are eight increments(1.0^oC) for each zone, and fourteen increments(0.5^oC) for storage tank to cover corresponding temperature range. Totally there are 9*9*15 = 1215 nodes for this three dimensional problem. With the initial condition $x(0) = x(24)$, there must be one or more closed optimal trajectories lying in this restricted state space.

The state increment dynamic programming method used here considers only node to node transitions, no interpolations in the state variables are allowed. If current node is x_a, it is allowed to transit to nearby node x_b which belongs to the set

$$x_b = x_a + k^t \triangle x \qquad k^t = [k_1\ k_2\ k_3] \qquad k_i \in (-1, 0, 1)$$

$$\triangle x = [\triangle x_1, \triangle x_2, \triangle x_3]^t \tag{3.19}$$

The stage variable time t ($0 \leq t \leq 24$) does not need to be quantized. Since the technique of label-tracing procedure in combination with state increment dynamic programming is used the optimal trajectory can been found directly. δt (transition time), the time interval over which a given control is applied, may be freely specified . If the maximum is $\triangle t_{max}$, the δt considered belong to the set

$$\triangle t_{max}/l \qquad l = 1, 2, ...l_{max} \tag{3.20}$$

The input vector v for achieving a particular transition in the set (3.19) with transition time in the set (3.20) is given by

$$v_i = k_i \triangle x_i / \triangle t - d_i = l k_i \triangle x_i / \triangle t_{max} - d_i \qquad i = 1, 2, 3 \tag{3.21}$$

The associated u vector which should satisfy the constraints $u_i \geq 0$ is then computed from Eq.(3.17). If the last update-time of the next node is t ($0 \leq t \leq 24$), the optimal control $\hat{u}(x_a, x_b, t - \delta t)$ can be found by comparing the trial costs

$$J_{trial}[x_a, t - \delta t, u(x_a, x_b, t - \delta t)] = l[x_a, u(x_a, x_b)]\delta t + I[x_b, t] \tag{3.22}$$

where $l[x_a, u(x_a, x_b)]$ is approximation of the change in the performance criterion Eq.(3.9) over the time interval form $(t - \delta t)$ to t and $I[x_b, t]$ is the minimum cost of the next node x_b (in forward sense of time). These trial costs cannot simply be compared directly, however, since different transitions may involve different transition time δt and in this case the trial cost pertain to different update times $t - \delta t$. A fair comparison requires the trial costs computed for different update times to be extrapolated in time, to a common time that is the earliest update time for the transitions considered. If a particular transition involves elapsed time δt and update time $t - \delta t$, the extrapolation formula is

$$J_{trial}[x_a, t_{min}, u] = J_{trial}[x_a, t-\delta t, u] + (J_{trial}[x_a, t-\delta t, u] - J[x_a, t])(t-\delta t-t_{min})/(\delta t) \tag{3.23}$$

where t_{min} is the earliest update time considered. Once these extrapolated costs have been computed, the least cost transition is selected as the optimal trajectory segment.

For each node x ($x \in X$), the minimum-cost trajectory segment, terminating on x and originating from a nearest-hypercube node, will be found, subsequently these candidate minimum-cost trajectories will be extended backwards in time node by node. If an optimal orbit lies in the state space, it will have the property that its initial node x(0) is the same as its terminal node x(24). In order to recognize this event, it is useful to maintain a "label table" for all nodes, an array of labels that is initialized at time $t = 24$ with the coordinates of each candidate terminal node. As each candidate minimum cost trajectory is extended backwards in time by finding a further segment, the label of the segment-terminating node is carried back to the segment originating node. Thus the initial node of an optimal orbit will be recognized as one which receives back its own original label after 24 hours of reversed time. Moreover, once this optimal initial node is known, the entire ensuing optimal trajectory can also be exhibited if the label table contains not just terminal-node coordinates, but complete strings of node-coordinates and update-times, for all the nodes encountered along a candidate trajectory. In this case, as each candidate trajectory is extended backwards in time by finding a further segment, the entire string of previously-encountered labels is carried back to the segment-originating node, and the coordinates of that node are affixed as the next element of the string.

In cases where there are not too many nodes in the state space, so the computational requirements refered to in Section 2.3.2 are not too great, this particular problem is simplified by treating the entire constrained state space as a single block so that block processing does not need to be considered. Then, with unknown initial condition but x(0)=x(24), every node in state space must be considered as a potential terminal state for an optimal orbit, with terminal cost $J[x,24]=0$ assigned. So at the terminal time $I(x,24)=0$ for all $x \in X$.

Once I(x,24) has been determined for x ($x \in X$), the next node to be updated is selected as the node in the state space which has the latest last update-time in label table (the process is moving in the reverse time direction, so the node selected can be thought of as being most in need of updating). The trial costs for the various admissible transitions are found as in Eq.(3.22), in general the associated update times are found by subtracting the transition times from the last update time of the target node. The trial costs must be extrapolated in time to a common earliest update time before the costs can be compared, according to Eq.(3.23). The minimum is selected as the optimal transition and the corresponding coordinates of the x (x$\in X$), the update time of the node and the control value of the optimal transition are saved in the label table. The procedure continues until all nodes x (x$\in X$) have been updated all the way back to t=0, and one or more of them is found to have been assigned its original label again.

3.3.2 Computation Requirement

For this particular problem, there are 1215 nodes in constrained state space. The procedure of state increment dynamic programming used here requires that sufficient data be stored to specify J_{trial} for every node at update time. In general, this is done by storing I(x,t) and its update time for each node in the state space. The number of locations of high-speed memory required here is then

$$N_h = 2 * 1215 * 8 = 19640 bytes \tag{3.24}$$

The maximum number of stages is approximately 120 so the number of values that should be retained is 120*2*1215 for finding the closed optimal trajectory. The totally low-speed memory is

$$N_c = 120 * 1215 * 2 * 8 = 2332800 bytes. \tag{3.25}$$

The results show that the computation requirement of this procedure is reasonable for storage capacity of the computer available in our department. A C program has been written for this procedure (Appendix A.1). The optimal result can be obtained by running the program about ten minutes on the work station.

3.3.3 Results and Conclusions

It is assumed that buildings are operated according to a schedule wherein the zone temperature are allowed to increase during unoccupied hours, e.g. setpoint or desired zone temperature at $24°C$ during occupied hours (8:00-18:00) and at $30°C$ during unoccupied hours (0:00-8:00) and (18:00-24:00). The energy price factor p(t) in the Eq.(3.9) is taken to be 1.0 for peak hours (8:00-24:00), and smaller value 0.5 for off-peak hours (0:00-8:00), which means that the price of the purchased energy at the off-peak hours is low and at the peak hours is high. Other parameters were specified in Table 3.1, for the typical cooling loads shown is Figure 3.2, with performance index weights specified as $R_1 = R_2 = 3 * 10^{-5}$ and cost parameter of zones temperature derivation $a_1 = a_2 = 1000$, the 24-hour optimal trajectory was computed by executing the C program. The results are shown in Figure 3.3. It appears that all states satisfy the periodic boundary

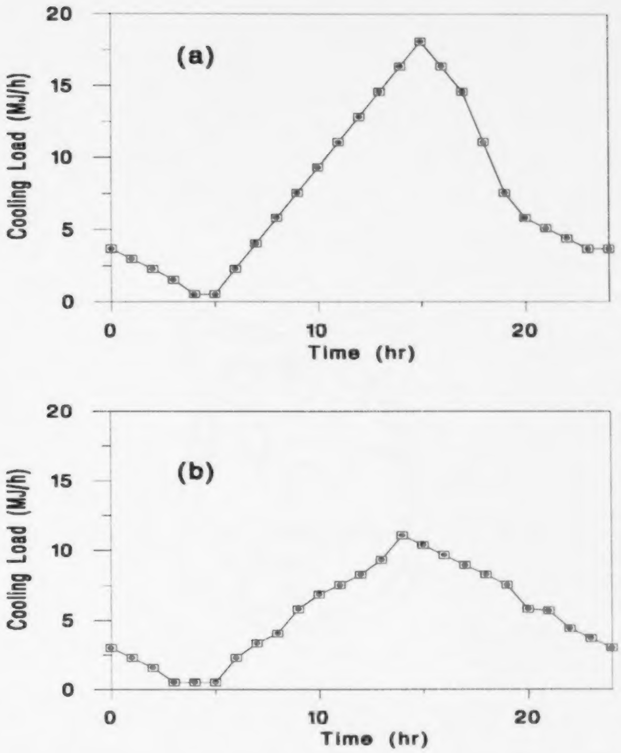


Figure 3.2: Typical daily cooling loads acting in the zones: (a) zone 1 load profile; (b) zone 2 load profile.

condition $x(0) = x(24)$ and also Figure 3.3 (b) shows that the temperature of the storage (T_c) is reduced to its minimum value just before 0800 hrs, while zone cooling loads are still low and energy price is low. This corresponds to charging the storage in anticipation of loads later in the day, and it is optimal to do so despite the reduction in coefficient of performance of the chiller when working against larger temperature differences. Note also that the zone temperatures (Figure 3.3 (a)) remain on the high side of setpoints during unoccupied hours (0:00-8:00) and (18:00-24:00), another indication of the minimization of purchased energy while respecting the comfort objectives.

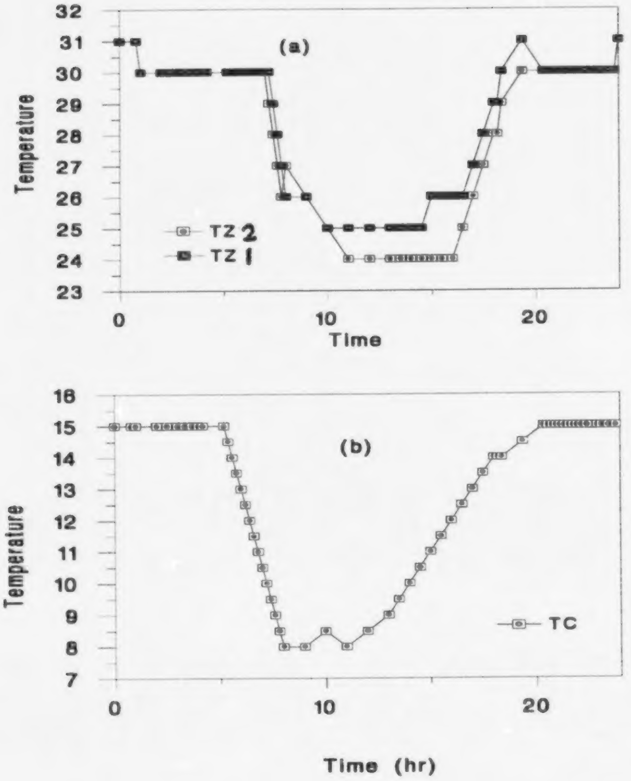


Figure 3.3: Optimal temperature profiles for two zones: (a) temperatures for zone 1 and zone 2; (b) temperatures for the storage tank.

Parameter	*Magnitude and unit*
a_{z1}	$410KJ/h^\circ C$
a_{z2}	$330KJ/h^\circ C$
a_c	$17.6KJ/h^\circ C$
ζ	$.75KJ/kg^\circ C$
C_{z1}	$374KJ/^\circ C$
C_{z2}	$300KJ/^\circ C$
C_c	$90000KJ/^\circ C$
COP_{max}	4.0
T_{max}	$20^\circ C$

Table 3.1: Parameters for two-zone cooling system

Another interesting application is showing the effect of the parameters of zones temperature derivation a_1 and a_2. The cooling loads of the first zone are kept unchanged and those of the second zone are made five times of previous value, which means that the cooling loads of the second zone are much heavier than first one, other parameters are same as above example except the capacity of the chiller changed to be $300000KJ/^\circ C$. The program is executed a few times with the value of $a_1 = a_2$ ranging from being small to large. In the Figure 3.4(b) it appears that the smaller the parameters a_1 and a_2 are the bigger the average temperature derivation and also the zone with heavier cooling loads has larger temperature derivation when the parameters a_1 and a_2 are not big enough to regulate the zones' temperature to setpoint. The Figure 3.4(a) shows that the larger parameters a_1 and a_2 are the more energy is consumed.

According to the results, we can conclude that parameters of the temperature derivation a_1 and a_2 affect significantly the optimal solutions for the multizone cooling system. The value of a_1 and a_2 should be chosen carefully, since too

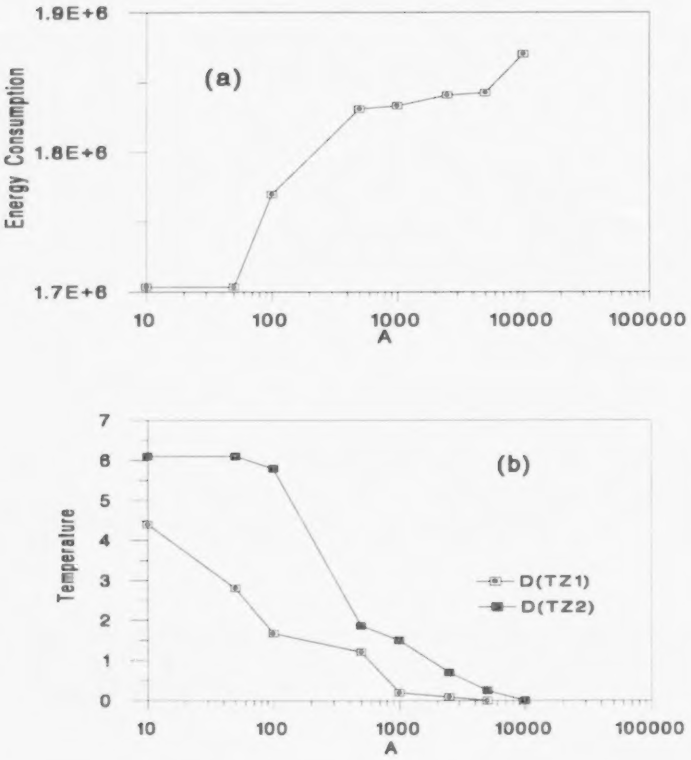


Figure 3.4: Optimal solution performance for two zones, energy consumption (a) and average temperature deviation from setpoint (b), vs. weight parameter A.

small values cause large zone temperature deviations and too big values cause more energy to be consumed. Also, the parameter of each zone can be different according to their cooling loads. Example results show that state increment dynamic programming, simplified through the use of feedback linearization and augmented with a label-tracing procedure, can provide an effective means for computing periodic optimal trajectory for controlling two-zone cooling systems. After solving two-zone cooling systems successfully, multizone cooling systems with many zones, hence many state variables, should be considered. Because of limited by high-speed memory of a computer, state increment dynamic programming can not be used directly. The methods of aggregation/disaggregation and block by block approach will be discussed in the following chapter.

Chapter 4

Aggregation and Disaggregation Method and Block by Block Approach

4.1 Multizone Cooling Systems With Many Zones

In this thesis, of particular interesting is the problem of finding the minimum-cost strategy for operating a multizone cooling system that might be typical of an office building divided into a number of zones, each with independent chilled-water supply from a common storage tank which in turn is chilled by a compression-cycle refrigeration unit. The components of multizone cooling systems with many zones are the same as those of two-zone cooling systems, the main difference is the number of zones. The multizone cooling systems considered here involved a maximum of eight zones. The diagram of the N-zone cooling systems is shown in Figure 2.1. The state equation of this system can also be written as

$$C_{zi}\dot{T}_{zi} = -u_i\zeta(T_{zi} - T_c) + a_{zi}(T_e - T_{zi}) + q_{zi}(t) \tag{4.1}$$

where $i = 1, 2, ..., N$.

$$C_c\dot{T}_c = -u_{N+1}COP + a_c(T_e - T_c) + \sum_{i=1}^{N} u_i\zeta(T_{zi} - T_c) \tag{4.2}$$

and the new state variables are defined by scaling the temperatures as

$$x_i = \sqrt{C_{zi}}T_{zi}; \qquad x_{N+1} = \sqrt{C_c}T_c \qquad for \;\; i = 1, 2, ..., N.$$

4.2 Computational Requirement

Before solving the problem of the multizone cooling systems, it is necessary to determine how much computational effort is required. Computational requirement includes the amount of memory needed to store both program and data, and the amount of the time required to perform the computations.

The most commonly encountered barrier to use of dynamic programming is the high-speed memory requirement. This requirement refers to the number of storages in the high-speed access memory (core memory) which must be available during computations. For the nine-dimension system with 8 quantization levels for each zone and 14 quantization for storage tank, directly using the state increment dynamic programming, according to previous analysis, the high-speed memory requirement is

$$N = 8 * 9^8 * 15bytes$$

Adding the memory reserved for program, compiler, and functions, the high speed memory of existing computer will be saturated. The second storage problem arises in retaining the results of the computation. If there are 100 stages during 24 hours, for this $9^8 * 15$ nodes, the low-speed memory requirement is

$$N_c = 100 * 8 * 9^8 * 15 bytes$$

This is also large and unfeasible number.

The computing time requirement is related to the number of the result obtained. If it takes $50 * 10^{-6}$ seconds for each result, then the total amount of computing time is

$$T_c = 50 * 10^{-6} * 100 * 9^8 * 15 secs$$

This number is more than one month.

From this analysis, directly using state increment dynamic programming to solve multizone cooling systems is not practical. The new approaches which fit this problem should be used.

4.3 Aggregation and Disaggregation Method

4.3.1 Principles of the Aggregation and Disaggregation Method

As the first step, the zones are initially aggregated into two macrozones, according to the 24 hour periodic is zone cooling load profiles, such that within each macrozone the differences are as small as possible. This is commonly possible in multizone cooling problems, where, e.g., a glass-wall office building can be considered to contain one group of sun-facing zones with high daytime cooling loads and another group of shaded zones with much lower and different cooling load profiles. Each of these group can reasonably be aggregated into one macrozone. After this aggregation has been done, this problem can be treated as a two-zone problem. The optimal solution can be found by application of the procedure described in the previous chapter. Once this problem has been solved, each macrozone can be disaggregated successively in binary fashion. At each stage a macrozone is divided into two smaller macrozones and the heat-removal rate previously computed for the larger macrozone is optimally distributed to the smaller two, again by an application of state increment dynamic programming. This disaggregation process continues until all the original zones are retrieved with projected heat-removal rates and zone temperature profiles. The overall result is suboptimal, since the initial optimization does not take into account the variations between the zones that are aggregated in a single macrozone. Nevertheless, we can show that the results are close to optimal if the groups of zones selected for aggregation are such that within group variations are as small as possible.

Aggregation of a group of zones is achieved by simply adding together their heat capacities C_{zi}, heat transfer coefficients a_i and cooling loads. For each group, there are N_j zones then

$$C_{ag_j} = \sum_{i=1}^{N_j} C_{z_i} \quad a_{ag_j} = \sum_{i=1}^{N_j} a_i \quad q_{ag_j}(t) = \sum_{i=1}^{N_j} q_{z_i}(t). \tag{4.3}$$

where

$$N = \sum_{j=1}^{2} N_j$$

The state equation for the two macrozones and storage tank can be written as

$$C_{ag_j}\dot{T}_{ag_j} = -u_{ag_j}\zeta(T_{ag_j} - T_c) + a_{ag_j}(T_e - T_{ag_j}) + q_{ag_j}(t) \tag{4.4}$$

$$C_c\dot{T}_c = -u_3COP + \sum_{j=1}^{2} u_{ag_j}\zeta(T_{ag_j} - T_c) + a_c(T_e - Tc) \tag{4.5}$$

where

$$j = 1, 2.$$

T_{ag_j} is temperature of macrozone j, the first term on the right-hand side of Eq.(4.4) is rate of heat transfer from macrozone j to circulating chilled water with pump rate u_{ag_j}, the second term is heat transfer rates from the environment to macrozone j, and the third term is the cooling loads generated by occupants and equipment within the macrozone j. Eq.(4.5) represents the heat balance for the chilled-water storage tank. The new state variables are defined by scaling the temperatures as

$$x_j = \sqrt{C_{ag_j}}T_{ag_j} \qquad x_3 = \sqrt{C_c}T_c$$

The pumping power for the aggregated zone is approximated as

$$R_{ag_j}u_{ag_j}^2 = \sum_{i=1}^{N_j} R_i[q_iu_{ag_j} / \sum_{i=1}^{N_j} q_i(t)]^2 \tag{4.6}$$

and hence

$$R_{ag_j} = \sum_{i=1}^{N_j} R_i[q_i / \sum_{i=1}^{N_j} q_i(t)]^2 \tag{4.7}$$

Under the same assumption of the Chapter 3, the optimal solution can be obtained with the procedure of state increment dynamic programming simplified through the use of state feedback linearization and decoupling, and augmented with a label-tracing method to identify period but unknow initial/terminal state by minimizing of

$$J = \int_0^{24} [p(t)u_c(t) + \sum_{j=1}^{2} [p(t) * R_{ag_j} u_{ag_j}^2 + A_j(T_{ag_j} - T_{s_i})]]dt \tag{4.8}$$

Once the optimum pumping rate $U_{ag_j}(t)$ and temperature $T_{ag_j}(t)$ of the resulting macrozone have been computed, its time integrated heat removal rate

$$I_{ag_j} = \int_0^t u_{ag_j}(\tau)\zeta[T_{ag_j}(\tau) - T_c(\tau)]d\tau \tag{4.9}$$

is also computed.

The binary disaggregation of this macrozone is next accomplished by separating it into two groups of zones, each group aggregated into a smaller macrozone with integrated heat-removal rate

$$I_{ag_{ji}} = \int_0^t u_{ag_{ji}}(\tau)\zeta[T_{ag_{ji}}(\tau) - T_c(\tau)]d\tau \tag{4.10}$$

where i=1,2.

The heat balance equation for these two smaller macrozones will be:

$$C_{ag_{ji}}\dot{T}_{ag_{ji}} = -u_{ag_{ji}}\zeta(T_{ag_{ji}} - T_c) + a_{ag_{ji}}(T_e - T_{ag_{ji}}) + q_{ag_{ji}} \tag{4.11}$$

In this equation, the 24 hour periodic value of T_c is already known from the previous aggregation procedure, but the equation of state are still bilinear, linear in the state variables and in control variables, but not jointly linear in both kinds of variables. In order to simplify the state increment dynamic programming solution of this two dimension problem, the method of exact linearization by state feedback will also be used here. If new state variables are defined by scaling the temperatures as

$$x_1 = \sqrt{C_{ag_{j1}}} T_{agj1}; \qquad x_2 = \sqrt{C_{ag_{j2}}} T_{agj2} \tag{4.12}$$

the corresponding state dynamic equation can be written as

$$\dot{x} = Ax + u_{ag_{j1}} B_1 x + u_{ag_{j2}} B_2 x + Cu + d(t) \tag{4.13}$$

where the control vector u is $[u_{ag_{j1}} u_{ag_{j2}}]^t$, A is diagonal matrix

$$A = diag(-a_{ag_{j1}}/C_{ag_{j1}}, -a_{ag_{j2}}/C_{ag_{j2}}) \tag{4.14}$$

the B matrices are the symmetric matrices

$$B_1 = b_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}; B_2 = b_2 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.15}$$

where $b_1 = -\zeta/C_{ag_{j1}}$, $b_2 = -\zeta/C_{ag_{j2}}$ and C and d(t) are given by

$$C = T_c \zeta \begin{bmatrix} 1/\sqrt{C_{ag_{j1}}} & 0 \\ 0 & 1/\sqrt{C_{ag_{j2}}} \end{bmatrix}; \tag{4.16}$$

$$d(t) = \begin{bmatrix} (a_{ag_{j1}} T_e + q_{ag_{j1}})/\sqrt{C_{ag_{j1}}} \\ (a_{ag_{j2}} T_e + q_{ag_{j2}})/\sqrt{C_{ag_{j2}}} \end{bmatrix} \tag{4.17}$$

Treating the two state variables as observable outputs, the method of exact linearization by state feedback can be easily applied. The result is obtained by writing the equation of state in the form

$$\dot{x} = Ax + E(x)u + d(t) \tag{4.18}$$

where

$$E(x) = [B_1 x + c_1 | B_2 x + c_2]$$

for any state x where E(x) is nonsingular, i. e. where

$$det E(x) = [b_1 + T_c \zeta / \sqrt{C_{ag_{j1}}}][b_2 + T_c \zeta / \sqrt{C_{ag_{j2}}}] \neq 0$$

then u can be expressed as

$$u = -E^{-1}(x)[Ax - v] \qquad (4.19)$$

which implies that the feedback system with new input vector v satisfies the linear, decoupled state equation

$$\dot{x} = v + d(t) \qquad (4.20)$$

Again this representation simplifies the dynamic programming solution of the two dimension optimization problem. Assuming that $d_{ag_{ji}}$ is given as a forecast, a set of v vectors can easily be achieved for any node transition, the corresponding u vectors found from Eq.(4.19) are substituted into the performance index

$$\int_0^{24} \sum_{i=1}^{2} [p(t) R_{ag_{ji}}(t) u^2_{ag_{ji}}(t) + A(T_{ag_{ji}} - T_s(t))^2] dt \qquad (4.21)$$

while the constraint

$$I_{ag_{j1}} + I_{ag_{j2}} \leq I_{ag_j} \qquad (4.22)$$

is required to be satisfied. The two terms on the left side of this equation are the time-integrated heat removal rates for the two disaggregated zones, the term of the right side of the equation is the time-integrated heat removal rate for aggregated the macrozone. The equation means the cooling effect actually consumed by the two disaggregated zones can not exceed the value specified for the aggregated macrozone. The costs are compared with each other. The optimal solution $u_{ag_{j1}}$, u_{agj2}, $T_{ag_{j1}}$, $T_{ag_{j2}}$ can be obtained. The binary disaggregation is continued, at each stage each macrozone disaggregated into two smaller ones, until the pumping rate u_i and temperature T_i for each original zone have been found.

4.3.2 Optimization by State Increment Dynamic Programming

The procedure of state increment dynamic programming is used in both aggregation part and disaggregation part. The version of state increment dynamic programming used in the first part is same as the one used for the three dimension problem in Chapter 3, which will not be repeated here. The focus of this section deals with the method of state increment dynamic programming used to solve the disaggregation problem.

During disaggregation, for the two smaller macrozones, the operating point and the range of the temperature has already been specified. Quantizing state variables, there are eight increments for each macrozone to cover corresponding temperature range. Totally there are 81 nodes for this two dimensional problem. With the initial condition $x(0) = x(24)$, there should be one or more closed optimal trajectores in this restricted state space.

Again, the state increment dynamic programming used here considers only node to node transitions. If current node is x_a, it is allowed to transit nearby node x_b which belongs to the set

$$x_b = x_a + k^t \triangle x \qquad k^t = [k_1 \; k_2] \quad k_i \in (-1, 0, 1) \;\; \triangle x = [\triangle x_1, \;\; \triangle x_2]^t \tag{4.23}$$

The stage variable time t ($0 \leq t \leq 24$) does not need to be quantized. Since the technique of label-tracing procedure in combination with state increment dynamic programming is used, the optimal trajectory can been found directly. δt (transition time), the time interval over which a given control is applied, may be freely specified . If the maximum is $\triangle t_{max}$, the δt considered belong to the set

$$\triangle t_{max}/l \qquad l = 1, 2, ..., l_{max} \tag{4.24}$$

The input vector v for achieving a particular transition in the set(4.23) with transition time in the set(4.24) is given by

$$v_i = k_i \triangle x_i / \triangle t - d_i = l k_i \triangle x_i / \triangle t_{max} - d_i \qquad i = 1, 2 \tag{4.25}$$

The associated u vector which should satisfy the constraints $u_i \geq 0$ is then computed from Eq.(4.19). If the constraint

$$I_{ag_{j1}} + I_{ag_{j2}} \leq I_{ag_j} \tag{4.26}$$

is satisfied, this means that this transition is allowed. If the last update-time of the next node is t ($0 \leq t \leq 24$), the trial cost is

$$J_{trial}[x_a, t - \delta t, u(x_a, x_b, t - \delta)] = l[x_a, u(x_a, x_b)]\delta t + I[x_b, t] \tag{4.27}$$

where $l[x_a, u(x_a, x_b)]$ is approximation of the change in the performance criterion Eq.(4.21) over the time interval form $(t - \delta t)$ to t and $I[x_b, t]$ is the minimum cost of the next node x_b (in forward sense of time). These trial costs cannot simply be compared directly, as mentioned in the previous Chapter, since different transitions may involve different transition time δt and in this case the trial cost pertain to different update times $t-\delta t$. A fair comparison requires the trial costs computed for different update times to be extrapolated in time, to a common time that is the earliest update time for the transitions considered. If a particular transition involves elapsed time δt and update time $t - \delta t$, the extrapolation formula is

$$J_{trial}[x_a, t_{min}, u] = J_{trial}[x_a, t-\delta t, u] + (J_{trial}[x_a, t-\delta t, u] - J[x_a, t])(t-\delta t-t_{min})/(\delta t) \tag{4.28}$$

where t_{min} is the earliest update time considered. Once these extrapolated costs have been computed, the least cost transition is selected as the optimal trajectory segment. For finding the minimum-cost trajectory, the procedure is same as the one mentioned in 3.3.1, which will not be repeated here.

4.3.3 Application of Aggregation and Disaggregation Method

In order to give a clear exposition of principal of the aggregation and disaggregation method, a three-zone cooling systems is treated as an example, and the energy cost of the near optimal solution obtained by the method of aggregation and disaggregation will be compared with that of the optimal solution. a second application will demonstrate the use of aggregation and disaggregation method to solve a eight-zone problem.

Three-zone Cooling System

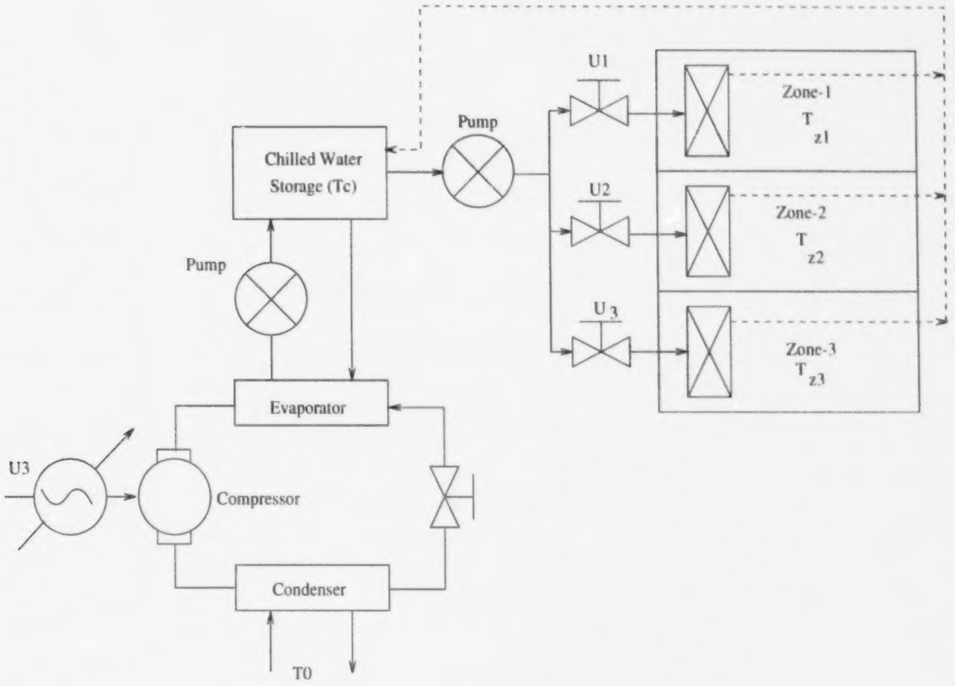

U1
Pump
Chilled Water
Storage (Tc)
Zone-1
T
z1
U2
Zone-2
T
z2
U 3
Zone-3
T
z3
Pump
Evaporator
U3
Compressor
Condenser
T0

Figure 4.1: Schematic diagram of a three-zone cooling system.

Z1	CZ1=374	AZ1=410	Load Profile (a) (Figure 3.2 (a))
Z2	CZ2=300	AZ2=330	Constant Load: 8.101 KJ/h
Z3	CZ3=100	AZ3=100	Constant Load: 4.223 KJ/h

Table 4.1: Parameters and Cooling Loads (CC=95000) for a three-zone system

For the three-zone problem, its schematic diagram is shown in Figure 4.1. The first zone, named zone1, is assumed to be a sun-facing zones with high daytime cooling loads and the other of two shaded zones, named zone2 and zone3, are assumed to have constant cooling load profiles. The paramenters and cooling loads of each zone are shown in Table 4.1. The three-zone cooling system is assumed to be operated according to the schedule wherein the zone temperatures are allowed to increase during unoccupied hours, e.g. with setpoint or desired zone temperature at $24^{\circ}C$ during occupied hours(8:00-18:00) and at 30°C during unoccupied hours (0:00-8:00) and (18:00-24:00). The energy price factor $p(t)$ is taken to be 1.0 for peak hours (8:00-24:00), and smaller value 0.5 for off-peak hours (0:00-8:00). In order to find the 24 hour suboptimal temperature trajectory, the aggregation and disaggregation method is used. According to their cooling-load profiles, the more similar two zones can be aggregated into one zone. In this case, the two shaded zones are aggregated into one macro zone named zag23 Considering zone1 and zag23 as two-zone cooling systems, the state equation can be written as

$$C_{z1}\dot{T}_{z1} = -u_{z1}\zeta(T_{z1} - T_c) + a_{z1}(T_e - T_{z1}) + q_{z1}(t) \tag{4.29}$$

$$C_{zag23}\dot{T}_{zag23} = -u_{zag23}\zeta(T_{zag23} - T_c) + a_{zag23}(T_e - T_{zag23}) + q_{zag23}(t) \tag{4.30}$$

$$C_c\dot{T}_c = -u_3COP + u_{z1}\zeta(T_{z1} - T_c) + u_{zag23}\zeta(T_{zag23} - T_c) + a_c(T_e - Tc) \tag{4.31}$$

where

$$C_{zag23} = C_{z2} + C_{z3}$$

$$a_{zag23} = a_{z2} + a_{z3}$$

$$q_{zag23}(t) = q_{z2}(t) + q_{z3}(t)$$

Using the C program for the two-zone cooling systems, the temperatures of the storage tank, zone1, and zag23 are obtained, the pumping rate for zone1 and zag23 are also obtained for this 24 hour period. Once these are obtained, the aggregated macrozone zag23 can be disaggregated. The problem to disaggregate the zag23 zone can be solved by minimizing the cost of energy consumed

$$J = \int_0^{24} [p(t)(R_2 u_{z2}^2 + R_3 u_{z3}^2) + A_2(T_{z2} - T_{2s})^2 + A_3(T_{z3} - T_{3s})^2] dt \qquad (4.32)$$

where the first two terms involving u_{zi} in the integrand represent power consumption by the circulation pumps, weighted by the price of energy p(t). The last two terms involving T_{zi} are weighted index of discomfort from zone temperature derivations away from setpoint. The steady-state solution of this problem will be developed by two-dimensional state increment dynamic programming where the associated control variable should satisfy (4.25), which can be written explicitly as

$$\int_0^t u_{zag23}(t)\zeta(T_{zag23} - T_c) dt \ \geq \ \int_0^t [u_{z2}(t)\zeta(T_{z2} - T_c) + u_{z3}(t)\zeta(T_{z3} - T_c)] dt \quad (4.33)$$

$$u_{zi} \geq 0 \quad for \ \ i = 2, 3.$$

The term on the left side ot the equation is the time-integrated heat removal rate for the *zag*23, the right side terms are the time-integrated heat removal rate for zone2 and zone3. The equation means the cooling effect actually consumed by z1

energy cost	off-peak hour	peak-hour	total
AggZ1,2	436476	313076	749552
AggZ1,3	439360	299981	739341
AggZ2,3	434977	303531	738508
optimal	388146	340697	728843

Table 4.2: Energy cost for three zone system

and z2 can not exceed the value specified for the $zag23$. A C program has been written for this two-dimension state increment dynamic programming. Executing this program, after the aggregation step, the 24-hour sub-optimal temperature and control sequences for the zone2 and zone3 are computed and the energy consumed by the whole system can also be obtained.

It is interesting to compare the results when the other two possible ways of forming the aggregated zone are considered, for this application. First, zone1 and zone2 are aggregated into one macrozone and zone3 is treated as an independent zone, then zone1 and zone3 are aggregated into one macrozone and zone2 is a single zone. The suboptimal temperature trajectores and total energy cost, for these two problem, are obtained by using the procedure mentioned above. The energy cost for these three different aggregations is shown in Table 4.2.

Using the procedure of Chapter 3 instead of this aggregation and disaggregation method to solve this three-zone problem, the optimal solution can be obtained and the optimal energy solution is also shown in Table 4.2. Comparing these results, we find that the method of Chapter 3 results in less energy consumption while aggregation, especially zone1 and zone2, procedure costs more energy. A conclusion can be drawn that the overall solution of the aggregation and disaggregation is suboptimal since the initial solution does not take into account the

Z1	CZ1=374	AZ1=410	Load Profile (a) (Figure 3.2 (a))
Z2	CZ2=370	AZ2=400	Load Profile (a) (Figure 3.2 (a))
Z3	CZ3=300	AZ3=330	Load Profile (b) (Figure 3.2 (b))
Z4	CZ4=187	AZ4=205	0.5 × Load Profile (a) (Figure 3.2 (a))
Z5	CZ5=374	AZ5=410	Constant Load: 8.101 KJ/h
Z6	CZ6=300	AZ6=330	Constant Load: 8.101 KJ/h
Z7	CZ7=187	AZ7=205	Constant Load: 5.223 KJ/h
Z8	CZ8=100	AZ8=110	Constant Load: 3.223 KJ/h

Table 4.3: Parameters and Nominal Cooling Loads (CC=250000) for eight-zone cooling system

within-group variations in zone loads and criteria, but may be close to optimal if these variations are small.

Eight-zone Cooling System

Directly using state increment dynamic programming is not feasible for the eight-zone cooling system shown in Figure 4.2 because of the large computation requirement. Results of the previous section show that the aggregation and disaggregation method can provide an effective means for computing periodic suboptimal trajectories for controlling multizone cooling systems. This method is used to solve this eight-zone problem. The eight-zone cooling system consists of four sun-facing zones with high daytime cooling load and four shaded zones with much lower and constant cooling loads. The parameters and cooling loads profiles of each zone are shown in Table 4.3. The system is assumed to be operated according to the schedule mentioned in previous section. To solve this problem, first the aggregation procedure should be done. According to their cooling-load

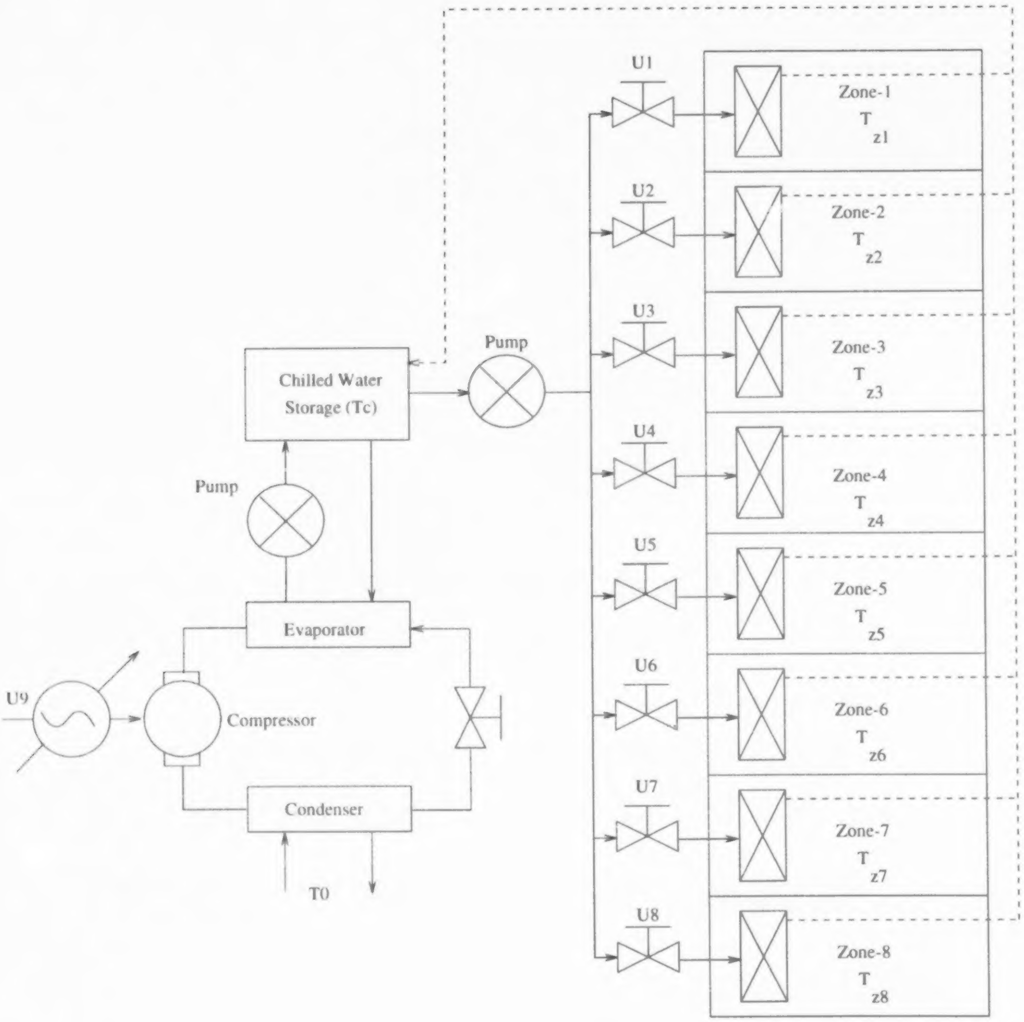

U1
U2
U3
U4
U5
U6
U7
U8
U9
Zone-1
T_{z1}
Zone-2
T_{z2}
Zone-3
T_{z3}
Zone-4
T_{z4}
Zone-5
T_{z5}
Zone-6
T_{z6}
Zone-7
T_{z7}
Zone-8
T_{z8}
Pump
Chilled Water Storage (Tc)
Pump
Evaporator
Compressor
Condenser
T0

Figure 4.2: Schematic diagram of a eight-zone cooling system.

profiles there are two groups: one is sun-facing, another is shaded. Each group is aggregated into one macrozone, where the sun-facing macrozone is named Zag1 and shaded one is Zag2. Considering these two macrozones as a two-zone cooling system, the pumping rate for each macrozone and the temperature for the storage tank and macrozone are obtained for the 24-hour period, after the two-zone optimal procedure described in Chapter 3 has been applied. The integrated heat-removal rate I_{ag1}, I_{ag2} is also calculated. After this has been done each macrozone will be successively disaggregated in binary fashion. For the macrozone Zag1, the four sun-facing zones are separated into two group zones according to the criterion that more similar zones are classified into the same group. In this case zone1 and zone2 are a group and zone3 and zone4 are another, each group forming a smaller macrozone with integrated heat-removal rate $I_{ag1_{12}}$ and $I_{ag1_{34}}$. Again using two-dimension state increment procedure the temperature and pumping rate are found by minimizing the cost

$$J = \int_0^{24} [p(t)(R_1 u^2_{ag1_{12}} + R_2 u^2_{ag1_{34}}) + A_1(T_{ag1_{12}} - T_{1s})^2 + A_2(T_{ag1_{34}} - T_{2s})^2] dt \quad (4.34)$$

while subject to the constraint

$$I_{ag1_{12}}(t) + I_{ag1_{34}}(t) \leq I_{ag1}(t) \quad (4.35)$$

After this has been done, we again use the two-dimension state increment procedure to disaggregate these two groups. The same procedure are applied to the shaded group, and finally the temperature trajectory for each zone is found. The result of the first zone and storage tank is shown in Figure 4.3. A C program has been written for this eight-zone aggregation/disaggregation method (Appendix A.2).

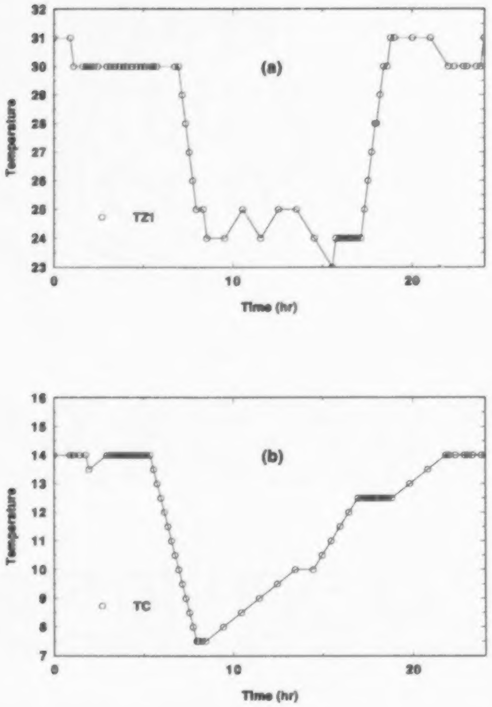


Figure 4.3: Optimal temperature profiles for eight zones: (a) temperatures for zone 1; (b) temperatures for the storage tank.

Table 4.4:

case	energy cost
(1) Suboptimal control, eight zones, nominal cooling loads	1904966
(2) Conventional PI control, constant reference temperature ($8°C$) for chilled storage, nominal cooling loads	3144267
(3) Temperature-tracking control, optimal for nominal cooling loads, with nominal cooling loads	1857585
(4) Temperature-tracking control, optimal for nominal cooling loads, with zone 1-4 cooling loads one-half of nominal	1424454
(5) Suboptimal control, eight zones for zone 1-4 cooling loads one-half of nominal	1351436
(6) Temperature-tracking control, optimal for nominal cooling loads, zone 1-4 cooling loads twice nominal	2744084
(7) Suboptimal control, eight zones for zone 1-4 cooling loads twice nominal	2707513

Table 4.4 summarizes results for an eight-zone study, parameters as shown Table 4.3, with energy costs for seven different cases. These cases were devised in order to examine how the performance (energy cost) changes when the controls and/or cooling loads depart from the ideal ones assumed in the aggregation/disaggregation computations.

Case (1) is the base case, where zone cooling loads are exactly as projected (the nominal cooling loads) and the storage and zone temperatures are exactly as computed, with chiller and circulation pumping rates determined by the precomputed solution. These assumptions are, of course, unrealistic in that

a) actual cooling loads experienced throughout the day will differ from the assumed nominal ones,

b) the state equations used for the dynamic programming solutions only approximate the thermal dynamics of the actual system, and

c) the assumption of periodicity for the optimization problem is not appropriate when the projected nominal cooling loads and/or environmental temperatures change from one day to the next.

For these reasons it would be necessary, in practice, to use a real-time, feedback control scheme to make actual temperatures track, in approximate fashion, temperature profiles that are precomputed in an "optimal" solution or that are simply specified as constant setpoints.

Case (2) represents a conventional way of controlling the system, wherein optimization is not used, but rather single-variable PI controllers are used to operate circulation pumps to cause the zone temperatures to track the setpoint temperature profile (Figure 4.4 (b)) and to operate the chiller in such a way as to cause storage temperature to track a constant ($8°C$) setpoint temperature (Fig. 4.4 (a)). The energy cost for this case is much larger than for case (1) because of the COP penalty in maintaining constant (low) chilled-storage temperature throughout the entire 24-hr. period. This energy cost dominates the energy savings implied by imperfect regulation of zone temperature by the PI controller during the period of high and rapidly changing cooling load, evident in (Figure 4.4 (a)). (Tighter regulation of zone temperature could be achieved with higher gains on the P and/or I controller terms, but with attendant loss of stability margin for the transient responses at the times of setpoint jumps.)

Case (3) represents the use of the same PI controller, but now used to cause

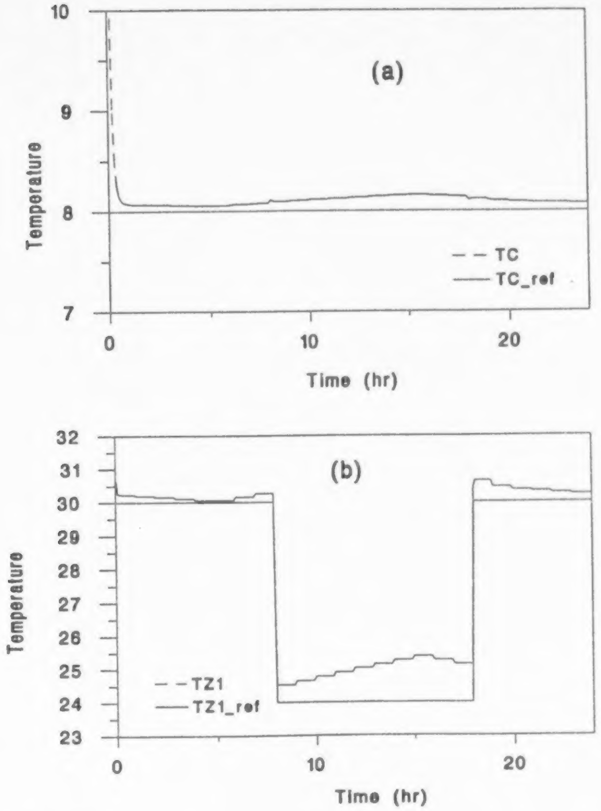


Figure 4.4: Tracking performance for storage temperature (a) and zone 1 temperature (b), PI controllers, piecewise-constant setpoints (reference temperatures). Parameters of table 4.3, case (2).

zone and storage temperatures to track the optimal profiles, precomputed for the nominal cooling loads. Here the energy cost is actually slightly lower than for case (1), due to the imperfect regulation of zone temperature (Figure 4.5 (b)).

Case (4) and (5) show the extent to which the operating efficiency of the system is degraded when some "actual" cooling loads are much lower than the forecasted loads upon which the suboptimal temperature trajectories are based. Case (5) shows the energy cost for the correct suboptimal solution, when half of the zones have forecasted loads one-half of nominal (e.g. zones with solar exposure, cloudy-

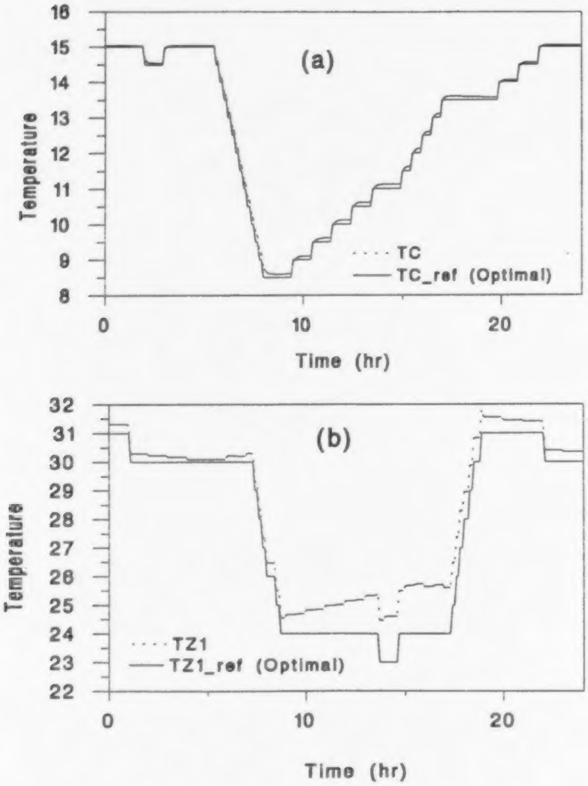


Figure 4.5: Tracking performance for storage temperature (a) and zone 1 temperature (b), PI controllers, reference temperatures from near-optimal solution. Parameters of table 4.3, case (3).

day forecast). Case (4) shows the energy cost incurred when the nominal (incorrect) suboptimal solution is tracked by the PI controllers throughout such a day. The latter cost is approximately 5% higher than the former, because the storage is pre-cooled to a lower temperature than it would optimally be for a day with lower cooling loads, hence lower-than-necessary COP values are experienced with resulting excess energy consumption. This amount of degradation is relatively small, considering the very large forecasting error involved.

Cases (6) and (7) are similar to (4) and (5), except that the "actual" cooling loads for zones 1-4 are twice nominal, i.e. a major forecasting error in the opposite direction. Case (6) shows that the performance degradation is less than 2% in this event. Here energy cost is increased because the storage is insufficiently charged, during the period of discounted energy price, for the cooling loads to come, and the chiller must be operated later in the day, while energy price is high. Again, the amount of efficiency degradation is relatively small.

The aggregation/disaggregation method has been shown to be an effective way of obtaining near-optimal solutions for the control of multi-zone HVAC systems with substantial energy storage, where a period of discounted energy price occurs daily. These solutions minimize energy cost by optimizing the pre-charging of the storage, according to the forecasted loads for the 24-hour period. Such solutions can be precomputed, off line, and stored in a high-capacity memory, from where the most appropriate one can be retrieved for the particular forecast each day. Real-time control would be implemented by conventional PI controllers, causing actual temperatures to track the precomputed optimal profiles.

4.4 Block by Block Method

4.4.1 Principles of The Block by Block Method

Another technique used to extending the dynamic programming method to multizone systems of much larger size is block by block method. The procedure described in the previous chapter computes optimal control at every admissible (quantized) state for all values of t. The larger the number of zones, the more high-speed memory and low speed memory will be required. These requirements grow exponential with the number of zones. For these optimization problems, when the number of zone is larger than four the procedure is unfeasible. The basic idea of the block by block approach as mentioned in chapter 2, is that it is not necessary to compute optimal control at all possible value of x and t. As a result, great savings in memory and computing time can be greatly achieved.

The block structure of state increment dynamic programming is ideally suited for restricting the computations so as to obtain only the results needed for the problem under consideration. The basic idea is to compute optimal control only in a region in which the optimal trajectory of interest is expected to lie. If such a region can be defined, then computations are performed only in those blocks contained in it. Clearly, if much information is available about this trajectory, then the region can be small, and hence a considerable reduction in both computing time and memory requirement can be obtained.

One method of determining the region in which computation take place is to use the aggregation approach mentioned in the previous section to calculate an approximation to the optimal trajectory, in a reduced-dimension state space. This could be done by aggregating all zones into two macrozones such that one is sun-facing and another is shaded. The state equation for these two macrozones can be written as:

$$C_{ag_j}\dot{T}_{ag_j} = -u_{ag_j}\zeta(T_{ag_j} - T_c) + a_{ag_j}(T_e - T_{ag_j}) + q_{ag_j}(t) \tag{4.36}$$

$$C_c\dot{T}_{cg} = -u_3COP + \sum_{j=1}^{2} u_{ag_j}\zeta(T_{ag_j} - T_c) + a_c(T_e - T_c) \tag{4.37}$$

where

$$j = 1, 2.$$

These equations are the same as in the previous section. The optimal temperature trajectory for the storage tank and macrozones can be obtained by using state increment dynamic programming directly. In order to achieve a more accurate optimal trajectory, the state variables should be quantized as small as possible during this procedure.

After the approximation to the optimal trajectory has been found the control variable are calculated only in the region of full-dimensional state space where the optimal trajectory is expected to lie in. This region can be defined by blocks which are (n+1)- dimensional rectangular sub-units representing the n state variable and time. Block size is four increment along each state dimension and the increment of time could be one or more, which depends on the temperature trajectories of the two microzones and storage. The initial time t_k is the previous blocks's end time and the end time t_{k+1} is time when any temperature of the two microzones or storage on the trajectory begins to change one increment. For example there is block B(1,2,3,...,M) contains values of Tc, Tz_i and t, such that

$$T_{cag} - 2 * \Delta T_c \leq T_c \leq T_{cag} + 2 * \Delta T_c \tag{4.38}$$

$$T_{ag_j} - 2 * \Delta Tz_i \leq Tz_i \leq T_{ag_j} + 2 * \Delta Tz_i \tag{4.39}$$

$$t_k \leq t \leq t_{k+1} \tag{4.40}$$

where

$$i = 1, 2, \ldots N - 1,$$

$$j = 1, 2$$

$$k = 1, 2, \ldots M$$

j should be 1 or 2 depends on which aggregation group T_{z_i} belongs to. As indicated by the above equations, the boundary between blocks is considered to be in both blocks. For a one dimensional example the blocks are two-dimensional rectangular unit shown in Figure 4.6 where each block contains 4 quantized values of T and time depends on the trajectory.

For obtaining the optimal control, the computations can take place block by block backwards in time. The state increment dynamic programming method, augmented with a label-tracing procedure to identify the periodic trajectory, as mentioned in previous chapter, is used for one block at a time on the assumption that optimal trajectories never leave the block. If the nodes on a boundary, which belongs to both blocks, already have been processed it is not necessary to recompute optimal control and minimum cost along this boundary. For the other nodes, the optimal control and minimum cost can be calculated node by node. If optimal trajectory lies in the set of blocks so treated, it will have the property that its initial node x(0) is the same as its terminal node x(24), after the blocks all have been processed. If there is no optimal trajectory the region

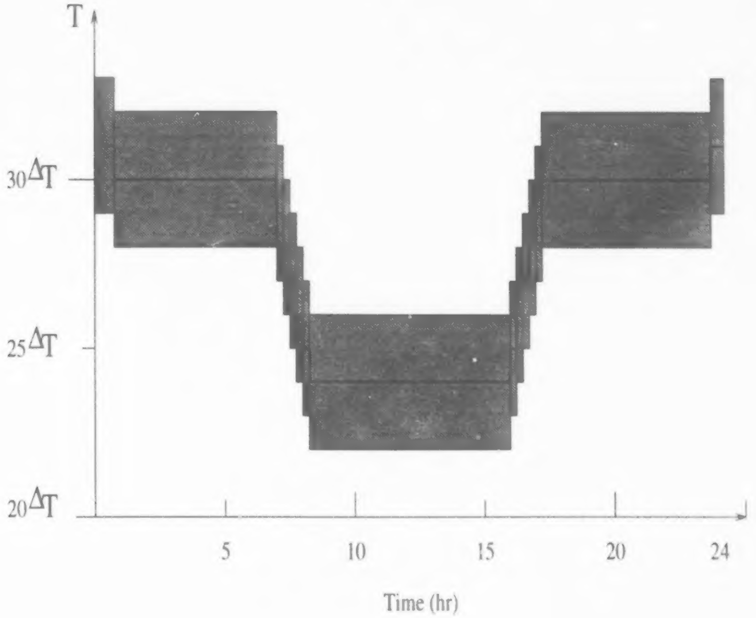


Figure 4.6: Determination of region from knowledge of an approximation to the optimal trajectory.

is too small. The region should be made larger and the blocks should be defined and calculated again.

4.5 Application of Block by Block Method

For the four-zone problem, with schematic diagram is shown in Figure 4.7, the first zone, named zone1, and the third zone, named zone3, are a sun-facing zones with high daytime cooling loads and the two shaded zones, named zone2 and zone4, have constant cooling load profiles. The paramenters and cooling loads of each zone are shown in Table 4.5. The four-zone cooling system is assumed to be operated according to the schedule wherein the zone temperatures are allowed to increase during unoccupied hours, e.g. with zone setpoint temperature at $24^{\circ}C$ during occupied hours(8:00-18:00) and at 30°C during unoccupied hours

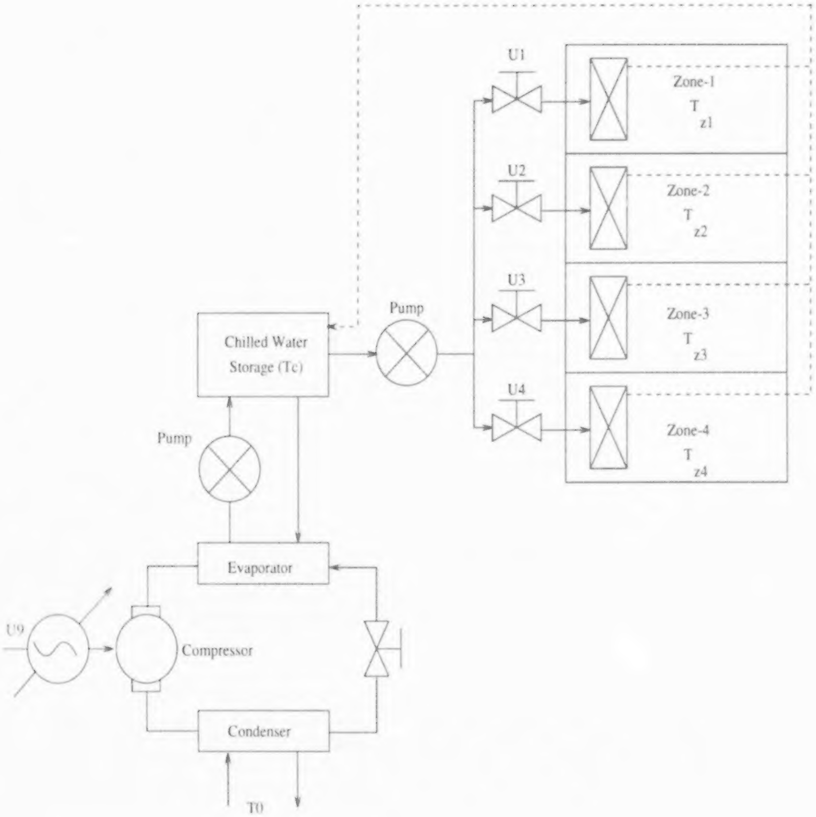

U1
Zone-1
T_{z1}
U2
Zone-2
T_{z2}
U3
Zone-3
T_{z3}
U4
Zone-4
T_{z4}
Pump
Chilled Water Storage (Tc)
Pump
Evaporator
U9
Compressor
Condenser
T0

Figure 4.7: Schematic diagram of a four-zone cooling system.

Z1	CZ1=374	AZ1=410	Load Profile (a) (Figure 3.2 (a))
Z2	CZ2=300	AZ2=330	Constant Load: 8.101 KJ/h
Z3	CZ3=187	AZ3=205	Half of Load Profile (a) (Figure 3.2 (a))
Z4	CZ4=187	AZ4=205	Constant Load: 5.223 KJ/h

Table 4.5: Parameters and Cooling Loads (CC=85000) for four-zone cooling system

(0:00-8:00) and (18:00-24:00). The energy price factor $p(t)$ is taken to be 1.0 for peak hours (8:00-24:00), and smaller value 0.5 for off-peak hours (0:00-8:00). In order to find 24-hour approximate temperature trajectory, the aggregation and disaggregation method is used. According to their cooling-load profiles, the more similar two zones can be aggregated into one macrozone, the two sun-faced zones are aggregated into one macrozone denoted zag13, and the two shaded zones are aggregated into one macro zone denoted zag24. Considering zag13 and zag24 as a two-zone cooling system, the state equation can be written as

$$C_{zag13}\dot{T}_{zag13} = -u_{zag13}\zeta(T_{zag13} - T_c) + a_{zag13}(T_e - T_{zag13}) + q_{zag13}(t) \tag{4.41}$$

$$C_{zag24}\dot{T}_{zag24} = -u_{zag24}\zeta(T_{zag24} - T_c) + a_{zag24}(T_e - T_{zag24}) + q_{zag24}(t) \tag{4.42}$$

$$C_c\dot{T}_c = -u_{3ag}COP + u_{zag13}\zeta(T_{zag13} - T_c) + u_{zag24}\zeta(T_{zag24} - T_c) + a_c(T_e - Tc) \tag{4.43}$$

where

$$C_{zag13} = C_{z1} + C_{z3}$$

$$a_{zag13} = a_{z1} + a_{z3}$$

$$q_{zag13}(t) = q_{z1}(t) + q_{z3}(t)$$

$$C_{zag24} = C_{z2} + C_{z4}$$

$$a_{zag24} = a_{z2} + a_{z4}$$

$$q_{zag24}(t) = q_{z2}(t) + q_{z4}(t)$$

Using the C program for the two-zone cooling systems, the temperatures of the storage tank, zag13 and zag24 are shown in Figure 4.8.

Figure 4.8 (b) shows that the storage temperature goes down just before beginning of the peak hour (8:00), this is optimal to store low priced energy to meet the rest of day's heavier loads. The interesting thing is that the temperature of the storage goes slightly down and then rises just before 18:00. It seems that the storage does not have quite enough capacity to store enough energy to meet the whole day's needs; the chiller must operate before 18:00 while energy cost is still higher and cooling must be maintained. Change the capacity of the storage to 90000 and then to 95000. Executing the program, the results shows that when the storage is CC = 90000 (see Figure 4.9), the temperature of the storage still goes down around 18:00, but not as low as CC = 85000 ; when CC = 95000 (see Figure 4.10), the temperature of storage goes down to lowest just before 8:00, then goes up smoothly, around 20:00 arrives at highest temperature and then keeps it during the rest of the day. From this application, it shows that not only the method but also parameters of the system affect the optimal solution.

After the approximation to the optimal trajectory has been found, the blocks can be defined according equations 4.37, 4.38, 4.39. A C program has been written for this block by block method (Appendix A.3). The result (CC = 85000) is shown in Figure 4.11.

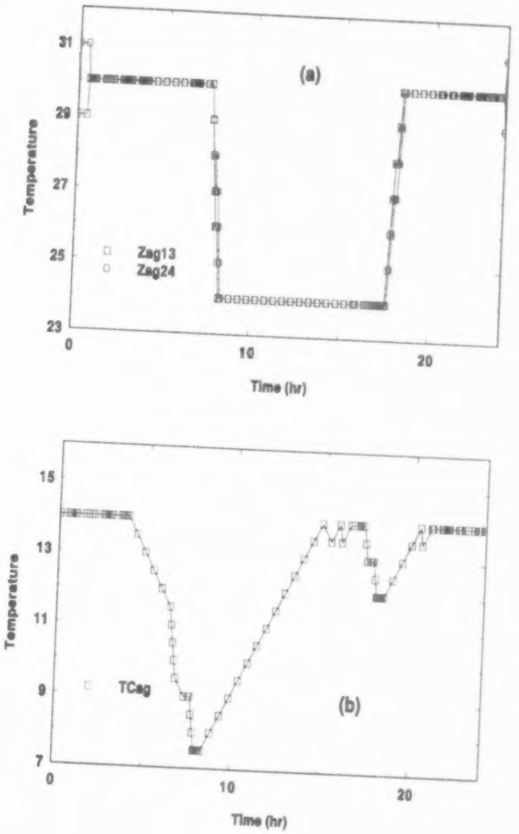


Figure 4.8: Optimal temperature profiles for two macrozones when CC = 85000: (a) temperatures for macrozone 1 and macrozone 2; (b) temperatures for the storage tank.

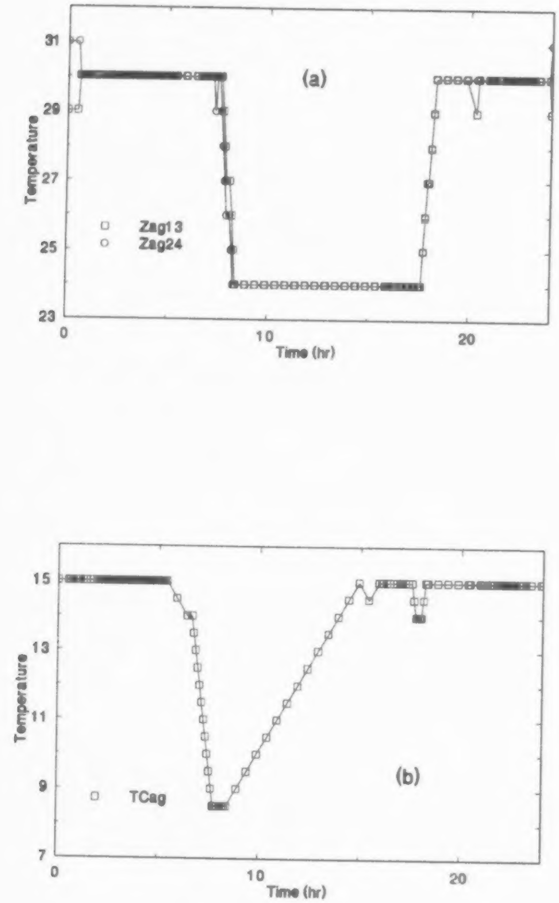
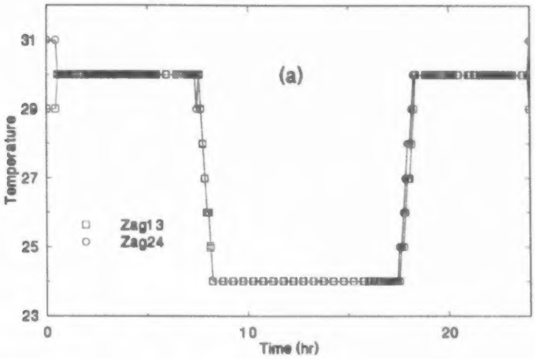


Figure 4.9: Optimal temperature profiles for two macrozones when CC = 90000: (a) temperatures for macrozone 1 and macrozone 2; (b) temperatures for the storage tank.



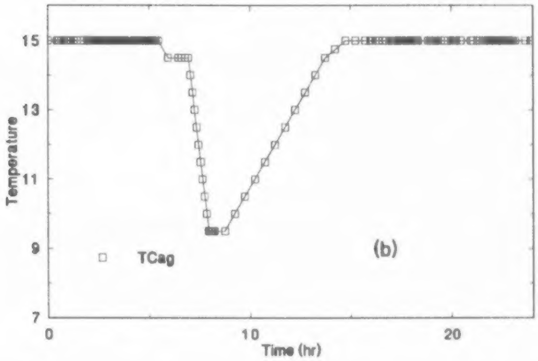


Figure 4.10: Optimal temperature profiles for two macrozones when CC = 95000: (a) temperatures for macrozone 1 and macrozone 2; (b) temperatures for the storage tank.

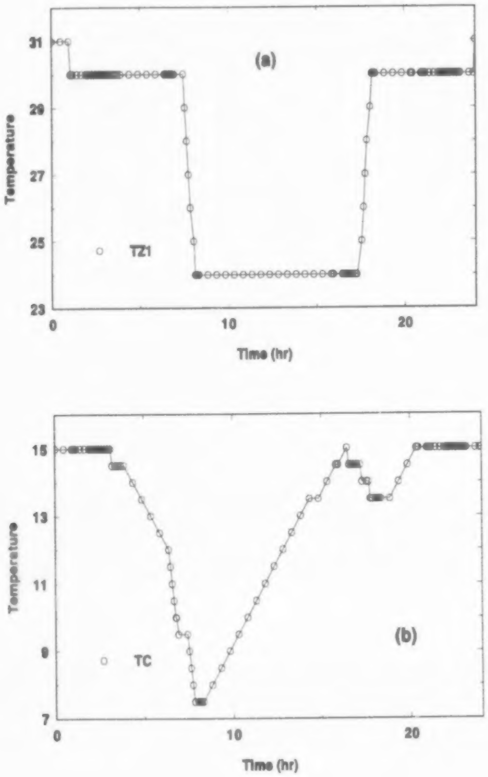


Figure 4.11: Optimal temperature profiles for four zones when CC = 85000: (a) temperatures for zone 1; (b) temperatures for the storage tank.

Although the solution of the block by block approach is optimal, the number of zones in a cooling system should not be too large. When the number is too large, the amount of high speend memory requirement can still saturate the now available computer. For example, if four increments for each state variable comprise one block, for the four-zone cooling system, the high speed requirement is 5*5*5*5*5*8 = 25000 bytes, this is reasonable value, but for a eight-zone cooling system, the requirement is 5*5*5*5*5*5*5*5*5*8 = 15625000 bytes, which is not feasible. The difficulty arises because, even though the blocks contain smaller volume than the original constrained region of state space, the number of nodes in each block is still very large when dimensionality is high.

Both the aggregation/disaggregation method and block by block approach have advantage and disadvantage. The aggregation method gives us sub-optimal solution, but it can be applied to cooling system with many zones, on the other hand, block by block approach gives us optimal solution, but the number of zones in a cooling system should not be unreasonable large.

Chapter 5

Conclusions

The purpose of this study is that of finding the minimum-cost strategy for operating a multizone cooling system that might be typical of an office building divided into a number of zones, each with independent chilled-water supply from a common storage tank which in turn is chilled by a compression-cycle refrigeration unit.

Of particular interest is the case where the storage capacity is substantial, a period of discount price occurs daily and a 24-hour projection of zone cooling loads are forecast. The operating optimal strategy is charging the storage during the period when the price of energy is low and retrieving energy for cooling when the price is high later in the day. The question under study is how much energy needs to be stored in order to minimize the cost and at the same time maintain zone temperatures within the comfort limits.

For the case which the number of zones is not larger than four, it has been shown that state increment dynamic programming, simplified through the use of feedback linearization and decoupling and augmented with a label-tracing procedure, can be used directly to computing periodic optimal trajectories.

A practical and reliable technique for extending the state increment dynamic programming method to multizone systems of much larger size is called aggregation and disaggregation method, wherein the most similar zones are initially aggregated into two macrozones. The three-dimensional periodic optimization problem is solved, then each macrozone is divided into two smaller macrozone and the heat-removal rate previously computed for the larger macrozone is optimally distributed to the smaller two, again by application of dynamic programming. This disaggregation process continues until all of the original zones are retrieved, now with projected heat-removal rates and zone temperature profiles. The overall result is suboptimal, since the initial optimization does not take into account the variations between the zones that are aggregated in a single macrozone. The example shows that nevertheless, the results are close to optimal if the groups of the zones selected for aggregation are such that within-group variations are as small as possible.

Another way for extending the state increment dynamic programming method to multizone systems of much larger size is block by block method. The basic idea is to compute optimal control only for a region in which the optimal trajectory of interest are expected to lie. If such a region can be defined, then computations are performed only in those blocks contained in it. Clearly, a considerable reduction in both computing time and memory requirement can be obtained. One method of determining the region in which computation take place is to use the aggregation approach to calculate an approximation to the optimal trajectory. The result of this method is optimal, but the number of the zones should not be unreasonably large since the large amount of high speed memory requirement can still saturate the computer.

Bibliography

[1] R. E. Rink and M. Zaheer-uddin. "Energy-Optimal Control of a Bilinear Multizone Cooling System with Chilled-water Storage", Energy Conversion and Management, Vol. 34, No. 12, pp.1229-1238, 1993.

[2] R. E. Rink, "Optimal Operation of Solar Heat Storage with Off-peak Energy Price Incentive", J. Optimization Theory and Applications.

[3] R. E. Rink, "Energy-Optimal Control of Bilinear Heat Storage Systems", Proc. 3rd Conference Communications and Control, Victoria, British Columbia, Canada, Oct. 16-18,1991.

[4] Rink, R.E., V. Gourishankar, M. Zaheer-uddin, "Optimal Control of Heat-Pump/Heat-Storge Systems with Time-of-Day Energy Price Incentive", J. Optimization Theory and Applications, Vol. 58, No. 1, pp. 93-108, 1988.

[5] Zaheer-uddin, M., R.E. Rink, V. Gourishankar, "A Design Criterion for a Solar-Assisted Heat Pump System", Energy, Vol. 12, No. 5, pp. 355-367, 1987.

[6] Le, H., M. Zaheer-uddin, V. Gourishankar, R.E. Rink, " Near-Optimal Control if a Bilinear, Solar-Assisted Heat Pump System", Tran. ASME, J. Solar Energy Engineering, Vol. 109, pp. 259-266 1987.

[7] Bellman R. "Dynamic Programming", Princeton University Press, Princeton, NJ, 1957.

[8] Larson, R. E., "Dynamic Programming with Reduce Computational Requirements", IEEE Trans. on Automatic Control, vol. AC-10, No.2, April, 1965, pp.135-143.

[9] R. E. Larson "State Increment Dynamic Programming", American Elsevier Publishing Company, INC. New York 1968.

[10] Shuttleworth, R., "Mechanical and Electrical Systems for Construction", McGraw-Hill, New York, 1983.

[11] Isidori, A., "Nonlinear Control Systems", Chapter 5, Second Edition, Springer-velag, Berlin, 1989.

[12] Bellman, R., and Dreyfus, S., "Applied Dynamic Programming", Princeton University Press, Princeton, NJ, 1962.

[13] Hamming, R. W., "Numerical Methods for Scientists and Engineers", McGraw-Hill Book Company, Inc., New York, 1962.

[14] Hildebrand, F. B., "Introduction to Numerical Analysis", McGraw-Hill Book Company, Inc., New York, 1956.

[15] Larson, R. E., "Dynamic Programming with Continuous Independent Variable", Stanford Electronics Laboratory TR 6302-6, Stanford, California, April 1964.

[16] Larson, R.E., "State Increment Dynamic Programming: Theory and Applications", Proc. 2nd Allerton Conf. on Ckt. and System Theory, U. of Illinois, Sept. pp. 643-665.

[17] Larson, R. E., "Dynamic Programming with Reduced Computational Requirements", IEEE Trans. on Automation Control, Vol. AC-10, No.2, April 1965, pp. 135-143.

[18] Larson, R. E., "An approach to Reducing the High-Speed Memory Requirement of Dynamic Programming", J. Math. and Appl., vol. 11, nos 1-3, July, 1965, pp. 519-537.

Appendix A

Source Code

A.1 Source Code for Two Zone Cooling System

```
/*********************************************************************************
***********
*
*       This is three dimensional state space optimization program. It supports
two_zone
*       cooling system. The optimal temperture trajectory of the zones and storage
tank
*       can be found after this program is executed.
*
**********************************************************************************
**********/

#include <stdio.h>
#include <math.h>

#define LENGTH(int)100                             /* max length of trajectories */
#define LM1    (int)(LENGTH-1)
#define LARGE  (double)1000000000.0
#define LP       (double)0.5                        /* off peak-hour energy price
ratio */
#defineHP       (double)1.0                         /* peak-hour energy price ratio */
#defineTZ1SLO (double)24.0                          /* setpoint temp. for occupied
zone */
#defineTZ1SHI (double)30.0                          /* setpoint temp. for unoccupied
zone */
#defineTZ2SLO (double)24.0
#defineTZ2SHI (double)30.0
#defineTZ1MIN (double)(TZ1SLO-1.0)                   /* min zone temperature constraint
*/
#defineIX1SPAN(int)9                               /* number of discrete points per
dim. */
#defineIX1MAX (int)(IX1SPAN-1)
#defineTZ1MAX (double)(TZ1SHI+1.0)                   /* max zone temperature constraint
*/
#defineTZ2MIN (double)(TZ2SLO-1.0)
#defineIX2SPAN(int)9
#defineIX2MAX (int)(IX2SPAN-1)
#defineTZ2MAX (double)(TZ2SHI+1.0)
#defineTCMIN  (double)8.0                          /* min tank temperature constraint
*/
#defineIX3SPAN(int)15
#defineIX3MAX (int)(IX3SPAN-1)
#defineTCMAX  (double)15.0                          /* max tank temperature constraint
*/
#defineT0     (double)15.0                         /* coolant temperature for
condenser */
#defineA1     (double)1000.0                       /* cost. parameter, zone temp.
deviation */
#defineA2     (double)1000.0
#defineR1     (double)0.00003                              /* cost. parameter,
circulation pump op. */
#defineR2     (double)0.00003
#defineR3     (double)1.0
#defineCZ1    (double)374.                         /* zone heat capacity, kw hours/deg
C */
#defineCZ2    (double)300.
#defineCC     (double)90000.                       /* tank heat capacity, kw hours/deg
C */
#defineG2     (double)CZ2/CC
#defineG1     (double)CZ1/CC
#defineAZ1    (double)410.0                        /* heat transfer coef. for zone */
#defineAZ2    (double)330.0
#defineAC     (double)17.6                         /* heat transfer coef. for tank */
#define ACZ   (double)0.75                         /* heat exchanger coef. */
#defineB1     (double)(-ACZ/CZ1)
#defineB2     (double)(-ACZ/CZ2)
#defineTMAX   (double)20.0                          /* max temp diff. refrig unit */
#defineCOPMAX (double)4.0                          /* coef. of performance, refrig
unit */
#defineCOPM1  (double)(COPMAX-1.0)
```

```
#defineBETA    (double)(COPM1/(CC*TMAX))
#defineC33     (double)(-COPM1*(1.0 - T0/TMAX)/RCC)
#defineB3      (double)(-BETA)
#defineDTMAX   (double)1.0                      /* max time-step allowed when time
is not smaller than -23.0 */
#defineLMAX    (int)5                           /* number of control amplitudes
tested */
#defineX1MIN   (double)(RCZ1*TZ1MIN)             /* min value, normalized state
variable */
#defineX1MAX   (double)(RCZ1*TZ1MAX)             /* max value, normalized state
variable */
#defineDX1     (double)((X1MAX-X1MIN)/IX1MAX)          /* stepsize, normalized
state var */
#defineX2MIN   (double)(RCZ2*TZ2MIN)
#defineX2MAX   (double)(RCZ2*TZ2MAX)
#defineDX2     (double)((X2MAX-X2MIN)/IX2MAX)
#defineX3MIN   (double)(RCC*TCMIN)
#defineX3MAX   (double)(RCC*TCMAX)
#defineDX3     (double)((X3MAX-X3MIN)/IX3MAX)
#defineX1S     (double)(RCZ1*tz1s)              /* desired value, normalized state
var */
#defineX2S     (double)(RCZ2*tz2s)
#definex1u     (double)(X1MIN + ix1u*DX1)        /* discrete level, node being
updated */
#definex2u     (double)(X2MIN + ix2u*DX2)
#definex3u     (double)(X3MIN + ix3u*DX3)
#defineg1      (double)(x1u-RG1*x3u)
#defineg2      (double)(x2u-RG2*x3u)
#defineg3      (double)x3u
#defined1      (double)((qz1+AZ1*35.0)/RCZ1)           /* normalized zone cooling
load */
#defined2      (double)((qz2+AZ2*35.0)/RCZ2)
#defined3      (double)(AC*35.0/RCC)
#define CLAMP(vv,ll,hh) ((vv)<(ll) ? (ll) :(vv)>(hh) ? (hh) :(vv))
#define ix1n    (CLAMP(ix1u+dix1, 0, IX1MAX))
#define ix2n    (CLAMP(ix2u+dix2, 0, IX2MAX))
#define ix3n    (CLAMP(ix3u+dix3, 0, IX3MAX))
#definex1n     (double)(X1MIN + ix1n*DX1)        /* discrete level, next node
examined */
#definex2n     (double)(X2MIN + ix2n*DX2)
#definex3n     (double)(X3MIN + ix3n*DX3)


/*
*       function declarations
*/

void main(void);
void test(void);
void test1(int);
void plota(int);
void plotb(void);
void plotu1(void);
void plotu2(void);
void plotu3(void);


/*
*       global variables
*/

double cotab[IX1SPAN][IX2SPAN][IX3SPAN];                /* table of running costs */
double  ectab[IX1SPAN][IX2SPAN][IX3SPAN][2];                   /* table of energy
costs */
double t1tab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH];         /* table of update times */
double u1tab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH];         /* table of u1 */
double u2tab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH];         /* table of u2 */
double u3tab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH];         /* table of u3 */
double  T[4][LENGTH];                                          /* table of
temperature */
double  COP[LM1];                                              /* table of COP */

int     labtab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH][3];           /* table of lable-
strings */
```

```
int     ix1u,ix2u,ix3u,ix1,ix2,ix3,m;
int     dix1,dix2,dix3,ix1next,ix2next,ix3next;
double u1,u2,u3,v1,v2,v3,RG1,RG2,RCZ1,RCZ2,RCC,qz1,qz2,tz1s,tz2s,dt_max;
double tn,tm,dt,t1,uu1,uu2,uu3,t1u,nco,ct,tc,ctc,ctcmin,r1,r2,r3,a1,a2,nec,price;

/*
*       lists of hourly zone cooling loads for zone 1.
*/
double qz1tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
             7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
              14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;

/*
*       lists of hourly zone cooling loads for zone 2.
*/
double qz2tab[25] = {2.987,2.275,1.572,0.517,0.517,0.517,2.275,3.330,4.034,
             5.792,6.847,7.550,8.254,9.309,11.067,10.364,9.661,
              8.957,8.254,7.550,5.792,5.088,4.385,3.682,2.987};

/*
*       lists of hourly enviroment temperture.
*/
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
             35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
             35.0,35.0};

/*
*        main program
*/

void main()
{
        double tlmax=0.0;                       /* start at midnight, go backwards (neg
time) */
        double  tmn,time;
        int     tnhour,tnhour1;
        int     l, n, first1, first2, first3, last1, last2, last3;


/*
*       initialize the arrays
*/

        RG1 = sqrt(G1);
        RG2 = sqrt(G2);
        RCZ1 = sqrt(CZ1);
        RCZ2 = sqrt(CZ2);
        RCC = sqrt(CC);

        for(n=0; n<4; n++)
         {       for(m=0; m<LENGTH; m++)
                {
                 T[n][m]=0.0;
                 COP[m]= 0.0;
                }
         }

        for(ix1=0; ix1<=IX1MAX; ix1++)
        {
                for(ix2=0; ix2<=IX2MAX; ix2++)
                {
                        for(ix3=0; ix3<=IX3MAX; ix3++)
                        {
                                cotab[ix1][ix2][ix3] = 0.0;
                                ectab[ix1][ix2][ix3][0] = 0.0;
                                ectab[ix1][ix2][ix3][1] = 0.0;
                                for(m=0; m<LENGTH; m++)
                                {
                                        t1tab[ix1][ix2][ix3][m] =
(rand()/32768.0)/1000000.0+0.0;
                                        u1tab[ix1][ix2][ix3][m] = 0.0;
                                        u2tab[ix1][ix2][ix3][m] = 0.0;
                                        u3tab[ix1][ix2][ix3][m] = 0.0;
                                        labtab[ix1][ix2][ix3][m][0] = ix1;  /* give each
node its own lable */
```

```
                                        labtab[ix1][ix2][ix3][m][1] = ix2;
                                        labtab[ix1][ix2][ix3][m][2] = ix3;
                                }
                        }
                }
        }
        while( tlmax > -24.0 ) {

/*
*       find the next node to update
*/

        tlmax = -24.0;
        for(ix1=0; ix1<=IX1MAX; ix1++)
        {
                for(ix2=0; ix2<=IX2MAX; ix2++)
                {
                        for(ix3=0; ix3<=IX3MAX; ix3++)
                        {
                                if(tltab[ix1][ix2][ix3][0]>tlmax)
                                {
                                        ix1u=ix1;            /*node to be updated has
latest last update*/
                                        ix2u=ix2;
                                        ix3u=ix3;
                                        tlmax=tltab[ix1][ix2][ix3][0];
                                        tlu = tlmax;
                                }
                        }
                }
        }

/*
*       find the earliest of the admissible update times
*/

        tm = 0.0;
        for(dix3=-1; dix3<=1; dix3++)
        {
                for(dix2=-1; dix2<=1; dix2++)
                {
                        for(dix1=-1; dix1<=1; dix1++)
                        {
                                tn=tltab[ix1n][ix2n][ix3n][0];
                                  tmn = tn-DTMAX;
                                if(tmn<=tm) tm=tmn;
                        }
                }
        }

/*
*       find the least-cost transition
*/

        tl = tltab[ix1u][ix2u][ix3u][0];
        ctcmin = LARGE;
        if((tl<-23.0) || ((tl<=-15.0) && (tl>-16.0)) ||((tl<=-5.0) && (tl>-6.0)))
        {

                tn=tl;
                dix1=0;
                dix2=0;
                dix3=0;
                price=LP;
                tnhour = tn+24;
                time = tn+24.0;
                tz1s=TZ1SHI;
                tz2s=TZ2SHI;
                r1=R1;
                r2=R2;
                r3=R3;
                a1=A1;
                a2=A2;
                qz1 = qz1tab[tnhour]*3600;
                qz2 = qz2tab[tnhour]*3600;
                dt_max=24.0+tn;

                 if ((tl<=-15.0) && (tl>-16.0))
                {
                        tz1s=TZ1SLO;
                         tz2s=TZ2SLO;
                         price=HP;
                         dt_max=16.0+tl;
                  }

                if ((tl<=-5.0) && (tl>-6.0))
                {
                         price=HP;
                          dt_max=6.0+tl;
                  }

                dt=dt_max;
                test1(l);

        }
        else
        {

                for(dix3=-1; dix3<=1; dix3++)
                {
                        for(dix2=-1; dix2<=1; dix2++)
                        [
                                for(dix1=-1; dix1<=1; dix1++)
                                {
                                        tn = tltab[ix1n][ix2n][ix3n][0];
                                        tnhour = tn+24;
                                        time = tn+24.0;
                                        qz1 = qz1tab[tnhour]*3600;
                                        qz2 = qz2tab[tnhour]*3600;
                                        tz1s=TZ1SHI;
                                        tz2s=TZ2SHI;
                                        r1=R1;
                                        r2=R2;
                                        r3=R3;
                                        a1=A1;
                                        a2=A2;
                                        if((time>8.0) && (time<=18.0))
                                        {
                                                tz1s=TZ1SLO;
                                                tz2s=TZ2SLO;
                                        }
                                        if(time<=8.0)
                                        price=LP;
                                        else
                                        price=HP;
                                        dt_max=DTMAX;

                                        for(l=LMAX; l>=1; l--)
                                        {
                                                dt=dt_max/l;
                                                if((time>=tnhour) && (tnhour>(time-dt)))
                                                [
                                                       tnhour1=24+tn-dt;
                                                       qz1=(qz1tab[tnhour1]*(tnhour-(time-
dt))/dt+qz1tab[tnhour]

                                                       *(time-tnhour)/dt)*3600;

                                                       qz2=(qz2tab[tnhour1]*(tnhour-(time-
dt))/dt+qz2tab[tnhour]

                                                       *(time-tnhour)/dt)*3600;

                                                      }
                                                test1(l);
                                        }
                                }
                        }
                ]
        }/* end of else */
        if(tlu >= tlmax)
        {
                printf("\failed update");
                        break;
        }
        for(n=LENGTH-2; n>=0; n--)
        {
                u1tab[ix1u][ix2u][ix3u][n+1] =
                        u1tab[ix1next][ix2next][ix3next][n];
                u2tab[ix1u][ix2u][ix3u][n+1] =
                        u2tab[ix1next][ix2next][ix3next][n];
                u3tab[ix1u][ix2u][ix3u][n+1] =
                        u3tab[ix1next][ix2next][ix3next][n];
                t1tab[ix1u][ix2u][ix3u][n+1] =
                        t1tab[ix1next][ix2next][ix3next][n];
                labtab[ix1u][ix2u][ix3u][n+1][0] =
                        labtab[ix1next][ix2next][ix3next][n][0];
                labtab[ix1u][ix2u][ix3u][n+1][1] =
                        labtab[ix1next][ix2next][ix3next][n][1];
                labtab[ix1u][ix2u][ix3u][n+1][2] =
                        labtab[ix1next][ix2next][ix3next][n][2];
        }/* end of for */
        cotab[ix1u][ix2u][ix3u] = nco;
        if (tlu<-16.0)
        {
                 ectab[ix1u][ix2u][ix3u][0] = nec+ectab[ix1next][ix2next][ix3next][0];
                 ectab[ix1u][ix2u][ix3u][1] = ectab[ix1next][ix2next][ix3next][1];
        }
        else
        {
                 ectab[ix1u][ix2u][ix3u][1] = nec+ectab[ix1next][ix2next][ix3next][1];
                 ectab[ix1u][ix2u][ix3u][0] = ectab[ix1next][ix2next][ix3next][0];
        }
         if(tlu==-6.0||tlu==-16.0)
                t1tab[ix1u][ix2u][ix3u][0] = tlu-(rand()/32768.0)/1000000.0;
         else
                  t1tab[ix1u][ix2u][ix3u][0] = tlu;

        u1tab[ix1u][ix2u][ix3u][0] = uu1;
        u2tab[ix1u][ix2u][ix3u][0] = uu2;
        u3tab[ix1u][ix2u][ix3u][0] = uu3;
        labtab[ix1u][ix2u][ix3u][0][0] = ix1u;
        labtab[ix1u][ix2u][ix3u][0][1] = ix2u;
        labtab[ix1u][ix2u][ix3u][0][2] = ix3u;


        } /* closes the "while" loop in main() */

        printf("\nLABLE TRACES");
        for(ix3=IX3MAX; ix3>=0; ix3--)
        {
                for(ix2=IX2MAX; ix2>=0; ix2--)
                {
                        for(ix1=0; ix1<=IX1MAX; ix1++)
                        {
                                for(n=0; n<LENGTH; n++)
                                {
                                        if(t1tab[ix1][ix2][ix3][n] <= -24.0)
                                        {
                                        first1 = labtab[ix1][ix2][ix3][0][0];
                                        last1 = labtab[ix1][ix2][ix3][LM1][0];
                                        first2 = labtab[ix1][ix2][ix3][0][1];
                                        last2 = labtab[ix1][ix2][ix3][LM1][1];
                                        first3 = labtab[ix1][ix2][ix3][0][2];
                                        last3 = labtab[ix1][ix2][ix3][LM1][2];
                                        if(first1==last1 && first2==last2 &&
first3==last3)
                                        
                                        {
                                        printf("\n");
                                        for(m=0; m<LENGTH; m++)
                                        {

        T[1][m]=(X1MIN+labtab[ix1][ix2][ix3][m][0]*DX1)/RCZ1;

        T[2][m]=(X2MIN+labtab[ix1][ix2][ix3][m][1]*DX2)/RCZ2;

        T[3][m]=(X3MIN+labtab[ix1][ix2][ix3][m][2]*DX3)/RCC;
```

```
                                           COP[m]=(COPMAX-1)*(1-(T0-T[3][m])/TMAX);
                                           printf("\nlabeltrace[%2d] = %2d,%2d,%2d",
m,

                                           labtab[ix1][ix2][ix3][m][0],
                                           labtab[ix1][ix2][ix3][m][1],
                                           labtab[ix1][ix2][ix3][m][2]);
                                           printf("\ttltrace[%2d] = %6f", m,
                                           tltab[ix1][ix2][ix3][m]);
                                    }/* end of for */
                                    printf("\npeak-hour energy cost = %12f",
                                    ectab[first1][first2][first3][1]);
                                    printf("\noff peak-hour energy cost = %12f",
                                    ectab[first1][first2][first3][0]);
                                        printf("\n");
                                    for(n=1; n<4; n++)
                                    {
                                           plota(n);
                                            printf("\n");
                                            for(m=0; m<LENGTH; m++)
                                            {
printf("\ntemtrace[%2d][%2d]=%6f",n,m,T[n][m]);
                                                    printf("\ttltrace[%2d] = %6f", m,
                                                    tltab[ix1][ix2][ix3][m]);
                                            } /* end of the for */
                                    }
                                        plotb();
                                    plotu1();
                                    plotu2();
                                    plotu3();
                                     printf("\n");
                                    for(m=0; m<LENGTH; m++)
                                    {
                                           printf("\nCOPtrace[%2d] =%6f",m,COP[m]);
                                            printf("\ttltrace[%2d]=%6f",m,
                                            tltab[ix1][ix2][ix3][m]);
                                    }
                                    } /* end of if */
                                    } /* end of the first if */
                             }
                      }
               }
        }
}/* end of the main function */


/*
*       procedure to call trial cost computation
*/

void test1(int l)
{
        v1 = (ix1n-ix1u)*DX1/dt - d1 + x1u*AZ1/CZ1;
        v2 = (ix2n-ix2u)*DX2/dt - d2 + x2u*AZ2/CZ2;
        v3 = (ix3n-ix3u)*DX3/dt - d3 + x3u*AC/CC;
        u1 = v1/(B1*g1);
        u2 = v2/(B2*g2);
        u3 = (RG1*v1 + RG2*v2 + v3)/(C33 + B3*g3);
        if((u1>=0.0) && (u2>=0.0) && (u3>=0.0))
        {
                test();
                return;
        }
        else
        {
                return;
        }
} /* end of test1 function */


/*
*       procedure to compute trial cost
*/
```

```
void test()
{
        double tltry;
        tltry=tn-dt;
        if((tltry<-6.0)&&(tl>-5.0))return;
        if((tltry<-16.0)&&(tl>-15.0))return;
        if((tltry<-24.0)&&(tl>-23.0))return;
        tc = ((rl*ul*ul + r2*u2*u2 + r3*u3)*price+ al*pow(xlu-X1S,2.0)/CZ1 +
                  a2*pow(x2u-X2S,2.0)/CZ2)*dt
               + cotab[ixln][ix2n][ix3n];
        ct = (tc - cotab[ixlu][ix2u][ix3u]) * (tn-dt-tm)/(tl-(tn-dt));
        ctc = tc + ct;
        if(ctc<=ctcmin)
        {
                ctcmin=ctc;
                nco=tc;
                tlu=tn-dt;
                uul=ul;
                uu2=u2;
                uu3=u3;
                nec=(rl*ul*ul + r2*u2*u2 + r3*u3)*dt ;
                ixlnext = ixln;
                ix2next = ix2n;
                ix3next = ix3n;
        }
        return;
} /* end of the test function */


/*
*       output time and temperature in plot files
*/
void plota(int n)
{
        FILE *fp, *fopen();
        if(n==1)
        fp=fopen("TZ1.dat","w");
        else if (n==2)
        fp=fopen("TZ2.dat","w");
        else
        fp=fopen("TC.dat","w");
        for(m=0; m<LENGTH; m++)
        {
                fprintf(fp,"\n%6f",T[n][m]);
                 fprintf(fp,"  %6f",(tltab[ixl][ix2][ix3][m]+24.0));
                if (tltab[ixl][ix2][ix3][m]>=0.0)
                        break;
         }
        fclose(fp);
        return;
} /* end of plota function */


/*
*       output time and COP in plotCOP file
*/
void plotb()
{
        FILE *fp, *fopen();
        fp=fopen("COP.dat","w");
        for(m=0; m<LENGTH; m++)
        {
                 fprintf(fp,"\n%6f",COP[m]);
                 fprintf(fp,"           %6f",(tltab[ixl][ix2][ix3][m]+24.0));
                if(tltab[ixl][ix2][ix3][m]>=0.0)
                        break;
         }
        fclose(fp);
        return;
} /* end of plotb function */

/*
*       output time and ul in plotUl file
*/
```

```
void plotu1()
{
        FILE *fp, *fopen();
        fp=fopen("U1.dat","w");
        for(m=0; m<LENGTH; m++)
        {
                fprintf(fp,"\n%6f",u1tab[ix1][ix2][ix3][m]);
                fprintf(fp,"           %6f",(t1tab[ix1][ix2][ix3][m]+24.0));
                if(t1tab[ix1][ix2][ix3][m]>=0.0)
                        break;
         }
        fclose(fp);
        return;
} /* end of the plotu1 */

/*
*       output time and u2 in plotU1 file
*/
void plotu2()
{
        FILE *fp, *fopen();
        fp=fopen("U2.dat","w");
        for(m=0; m<LENGTH; m++)
        {
                fprintf(fp,"\n%6f",u2tab[ix1][ix2][ix3][m]);
                 fprintf(fp,"           %6f",(t1tab[ix1][ix2][ix3][m]+24.0));
                 if(t1tab[ix1][ix2][ix3][m]>=0.0)
                break;
         }
        fclose(fp);
        return;
} /* end of the plotu2 */

/*
*       output time and u3 in plotU1 file
*/
void plotu3()
{
        FILE *fp, *fopen();
        fp=fopen("U3.dat","w");
        for(m=0; m<LENGTH; m++)
        {
                fprintf(fp,"\n%6f",u3tab[ix1][ix2][ix3][m]);
                 fprintf(fp,"           %6f",(t1tab[ix1][ix2][ix3][m]+24.0));
                 if(t1tab[ix1][ix2][ix3][m]>=0.0)
                        break;
         }
        fclose(fp);
        return;
} /* end of function plotu3 */
```

A.2 Source Code for Eight-Zone Cooling System

```
# This is script file for exacute the program for eight-zone cooling system

gcc -ansi -O -c agg8.c
gcc agg8.o  -o agg8 -lm


cp dagg11.h dagg.h
gcc -ansi -O -c dagg4.c
gcc -ansi -O -c dinp11.c
gcc dagg4.o dinp11.o -o dagg11 -lm
dagg11

cp dagg12.h dagg.h
gcc -ansi -O -c dagg4.c
gcc -ansi -O -c dinp12.c
gcc dagg8.o dinp12.o -o dagg12 -lm
dagg14

cp dagg21.h dagg.h
gcc -ansi -O -c dagg2.c
gcc -ansi -O -c dinp21.c
gcc dagg2.o dinp21.o -o dagg21 -lm
dagg21

cp dagg22.h dagg.h
gcc -ansi -O -c dagg2.c
gcc -ansi -O -c dinp22.c
gcc dagg2.o dinp22.o -o dagg22 -lm
dagg22

cp dagg23.h dagg.h
gcc -ansi -O -c dagg2.c
gcc -ansi -O -c dinp23.c
gcc dagg2.o dinp23.o -o dagg23 -lm
dagg23

cp dagg24.h dagg.h
gcc -ansi -O -c dagg2.c
gcc -ansi -O -c dinp24.c
gcc dagg2.o dinp24.o -o dagg24 -lm
dagg24
```

```
/***********************************************************************************
******
*
*      File Name:    agg8.h
*      Purpose:      This header file for eight-zone aggregation. It declares the
*                    variables and functions required by the file agg8.c
*
***********************************************************************************
******/

#include <stdio.h>
#include <math.h>

#define LENGTH(int)100                           /*max length of trajectories*/
#defineLM1    (int)(LENGTH-1)
#define LARGE (double)1000000000.0
#defineLP      (double)0.5                        /*off peak-hour energy price
ratio*/
#defineHP      (double)1.0                        /* peak-hour energy price ratio*/
#defineTZ1SLO (double)24.0                        /*setpoint temp for occupied
zone*/
```

```
#defineTZ1SHI  (double)30.0                            /*setpoint temp for unoccupied
zone*/
#defineTZ2SLO  (double)24.0
#defineTZ2SHI  (double)30.0
#defineTZ1MIN  (double)(TZ1SLO-1.0)                     /*min zone temperature
constraint*/
#defineIX1SPAN(int)9                                   /*number of discrete points per
dim.*/
#defineIX1MAX  (int)(IX1SPAN-1)
#defineTZ1MAX  (double)(TZ1SHI+1.0)                     /*max zone temperature
constraint*/
#defineTZ2MIN  (double)(TZ2SLO-1.0)
#defineIX2SPAN(int)9
#defineIX2MAX  (int)(IX2SPAN-1)
#defineTZ2MAX  (double)(TZ2SHI+1.0)
#defineTCMIN   (double)8.0                             /*min tank temperature constraint*/
#defineIX3SPAN(int)15
#defineIX3MAX  (int)(IX3SPAN-1)
#defineTCMAX   (double)15.0                             /*max tank temperature
constraint*/
#defineT0      (double)(35.0-20.0)                             /*coolant temperature for
condenser*/
#defineA1      (double)12000.0                                 /*cost parameter, zone temp
deviation*/
#defineA2      (double)12000.0
#defineR1      (double)0.00003                                  /*cost parameter,
circulation pump op.*/
#defineR2      (double)0.00003
#defineR3      (double)1.0
#defineCZR1    (double)(374.0)                                 /*zone heat capacity, kw
hours/deg C*/
#defineCZR2    (double)(370.0)
#defineCZR3    (double)(300.0)
#defineCZR4    (double)(187.0)
#defineCZR5    (double)(374.0)
#defineCZR6    (double)(300.0)
#defineCZR7    (double)(187.0)
#defineCZR8    (double)(100.0)
#defineCZ1     (double)(CZR1+CZR2+CZR3+CZR4)                   /*aggregated capacity*/

#defineCZ2     (double)(CZR5+CZR6+CZR7+CZR8)
#defineCC      (double)250000.                                 /*tank heat capacity, kw
hours/deg C*/
#defineG2      (double)CZ2/CC
#defineG1      (double)CZ1/CC
#defineAZR1    (double)(410.0)
#defineAZR2    (double)(400.0)
#defineAZR3    (double)(330.0)
#defineAZR4    (double)(205.0)
#defineAZR5    (double)(410.0)
#defineAZR6    (double)(330.0)
#defineAZR7    (double)(205.0)
#defineAZR8    (double)(110.0)
#defineAZ1     (double)(AZR1+AZR2+AZR3+AZR4)                   /*heat transfer coef for
aggregated macrozone*/
#defineAZ2     (double)(AZR5+AZR6+AZR7+AZR8)
#defineAC      (double)17.6                            /*heat transfer coef, tank*/
#define ACZ    (double)0.75                            /*heat exchanger coef*/
#defineB1      (double)(-ACZ/CZ1)
#defineB2      (double)(-ACZ/CZ2)
#defineTMAX    (double)20.0                             /*max temp diff, refrig unit*/
#defineCOPMAX  (double)4.0                             /*coef of performance, refrig
unit*/
#defineCOPM1   (double)(COPMAX-1.0)
#defineBETA    (double)(COPM1/(CC*TMAX))
#defineC33     (double)(-COPM1*(1.0 - T0/TMAX)/RCC)
#defineB3      (double)(-BETA)
#defineDTMAX1  (double)1.0                             /*max time-step allowed when time
is not smaller than -23.0*/
#defineLMAX    (int)5                                  /*number of control amplitudes
tested*/
#defineX1MIN   (double)(RCZ1*TZ1MIN)                     /*min value, normalized state
variable*/
#defineX1MAX   (double)(RCZ1*TZ1MAX)                     /*max value, normalized state
variable*/
```

```
#defineDX1      (double)((X1MAX-X1MIN)/IX1MAX)             /*stepsize, normalized
state var*/
#defineX2MIN    (double)(RCZ2*TZ2MIN)
#defineX2MAX    (double)(RCZ2*TZ2MAX)
#defineDX2      (double)((X2MAX-X2MIN)/IX2MAX)
#defineX3MIN    (double)(RCC*TCMIN)
#defineX3MAX    (double)(RCC*TCMAX)
#defineDX3      (double)((X3MAX-X3MIN)/IX3MAX)
#defineX1S      (double)(RCZ1*tz1s)                /*desired value, normalized state
var*/
#defineX2S      (double)(RCZ2*tz2s)
#definex1u      (double)(X1MIN + ix1u*DX1)          /*discrete level, node being
updated*/
#definex2u      (double)(X2MIN + ix2u*DX2)
#definex3u      (double)(X3MIN + ix3u*DX3)
#defineg1       (double)(x1u-RG1*x3u)
#defineg2       (double)(x2u-RG2*x3u)
#defineg3       (double)x3u
#defined1       (double)((qz1+AZ1*35.0)/RCZ1)              /*normalized zone cooling
load*/
#defined2       (double)((qz2+AZ2*35.0)/RCZ2)
#defined3       (double)(AC*35.0/RCC)
#define ix1n    (int)fix1n()                        /*index of next node examined*/
#defineix2n     (int)fix2n()
#defineix3n     (int)fix3n()
#definex1n      (double)(X1MIN + ix1n*DX1)          /*discrete level, next node
examined*/
#definex2n      (double)(X2MIN + ix2n*DX2)
#definex3n      (double)(X3MIN + ix3n*DX3)


/*********** function declarations*************/
void main( void);
int  fix1n(void);
int  fix2n(void);
int  fix3n(void);
void test(void);
void test1(int);
void plot(void);


/************ global variables ****************/

double cotab[IX1SPAN][IX2SPAN][IX3SPAN];                   /*table of running costs*/
double chtab[IX1SPAN][IX2SPAN][IX3SPAN];                   /*table of running costs*/
double  ectab[IX1SPAN][IX2SPAN][IX3SPAN][2];                     /*table of energy
costs*/
double t1tab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH];           /*table of update times*/
double u1tab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH];           /*table of u1*/
double u2tab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH];           /*table of u2*/
double u3tab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH];           /*table of u3*/
double  T[4][LENGTH];                                            /*table of
temperature*/
double  COP[LM1];                                                /*table of COP*/

int     labtab[IX1SPAN][IX2SPAN][IX3SPAN][LENGTH][3];            /*table of lable-
strings*/
int     ix1u,ix2u,ix3u,ix1,ix2,ix3,m,count;
int     dix1,dix2,dix3,ix1next,ix2next,ix3next;
double u1,u2,u3,v1,v2,v3,RG1,RG2,RCZ1,RCZ2,RCC,qz1,qz2,tz1s,tz2s,DTMAX;
double  pu1,pu2,pu3,pu4,pu5,pu6,pu7,pu8, pqz1,pqz2,pqz3,pqz4,pqz5, pqz6, pqz7,pqz8;
double tn,tm,dt,t1,uu1,uu2,uu3,t1u,nco,ctcmin,r1,r2,r3,a1,a2,nchill,nec,price,TE;
/*lists of hourly zone cooling loads*/
double qz1tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
             7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
              14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;
double qz2tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
             7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
              14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;
double qz3tab[25] = {2.987,2.275,1.572,0.517,0.517,0.517,2.275,3.330,4.034,
             5.792,6.847,7.550,8.254,9.309,11.067,10.364,9.661,
              8.957,8.254,7.550,5.792,5.088,4.385,3.682,2.987} ;
double  qz4tab[25] = {1.841,1.489,1.127,0.758,0.208,0.208,1.237,2.017,2.846,
             3.770,4.604,05.533,06.413,07.294,08.171,09.050,08.171,
             07.292,05.533,3.770,2.891,2.544,2.192,1.841,1.841};
```

```
double qz5tab[25] = {8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
              8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101} ;
double qz6tab[25] = {8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
              8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101} ;
double  qz7tab[25] = {5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
              5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
              5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223};
double  qz8tab[25] = {3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,
              3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,
              3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223};
/*lists of hourly envioment temperture*/
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
              35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
              35.0,35.0};




/*************************************************************************************
*********
*
*      File Name:     agg8.c
*      Purpose:       This program computes the optimal trajectory for two macrozones
*                     and storage. The main function calls module:
*                            fix1n()
*                            fix2n()
*                            fix3n()
*                            test()
*                            test1(int)
*                            plot()
*
**************************************************************************************
*******/

/*********** main program ******************/

#include<stdio.h>
#include "agg8.h"

void main()
{
       double tlmax=0.0;                   /*start at midnight, go backwards (neg
time)*/
       double  tmn,time;
       int     tnhour,tnhour1;
       int     1, n, first1, first2, first3, last1, last2, last3;


       /**** initialize the arrays *****/

       RG1 = sqrt(G1);
       RG2 = sqrt(G2);
       RCZ1 = sqrt(CZ1);
       RCZ2 = sqrt(CZ2);
       RCC = sqrt(CC);
       for(n=0; n<4; n++)
        {       for(m=0; m<LENGTH; m++)
               {
                T[n][m]=0.0;
                COP[m]= 0.0;
                }
        }
       for(ix1=0; ix1<=IX1MAX; ix1++)
       {      for(ix2=0; ix2<=IX2MAX; ix2++)
              {
                     for(ix3=0; ix3<=IX3MAX; ix3++)
                     {
                            cotab[ix1][ix2][ix3] = 0.0;
                            chtab[ix1][ix2][ix3] = 0.0;
                            ectab[ix1][ix2][ix3][0] = 0.0;
```

```
                                ectab[ix1][ix2][ix3][1] = 0.0;
                                for(m=0; m<LENGTH; m++)
                                {
                                        tltab[ix1][ix2][ix3][m] =
0.0+(rand()/32768.0)/1000000;
                                        u1tab[ix1][ix2][ix3][m] = 0.0;
                                        u2tab[ix1][ix2][ix3][m] = 0.0;
                                        u3tab[ix1][ix2][ix3][m] = 0.0;
                                        labtab[ix1][ix2][ix3][m][0] = ix1;  /*give each
node its own lable*/
                                        labtab[ix1][ix2][ix3][m][1] = ix2;
                                        labtab[ix1][ix2][ix3][m][2] = ix3;
                                }
                        }
                }
        }
        while( tlmax > -24.0 )
{

        /**** find the next node to update ****/

        tlmax = -24.0;
        for(ix1=0; ix1<=IX1MAX; ix1++)
        {
                for(ix2=0; ix2<=IX2MAX; ix2++)
                {
                        for(ix3=0; ix3<=IX3MAX; ix3++)
                        {
                                if(tltab[ix1][ix2][ix3][0]>tlmax)
                                {       ix1u=ix1;       /*node to be updated has
latest last update*/
                                        ix2u=ix2;
                                        ix3u=ix3;
                                        tlmax=tltab[ix1][ix2][ix3][0];
                                        tlu = tlmax;
                                }
                        }
                }
        }

        /**** find the earliest of the admissible update times*****/

        tm = 0.0;
                for(dix3=-1; dix3<=1; dix3++)
                {
                        for(dix2=-1; dix2<=1; dix2++)
                        {
                                for(dix1=-1; dix1<=1; dix1++)

                                        {tn=tltab[ix1n][ix2n][ix3n][0];
                                           tmn = tn-DTMAX1;
                                         if(tmn<=tm) tm=tmn;
                                }
                        }
                }

        /*** find the least-cost transition ***/

                tl = tltab[ix1u][ix2u][ix3u][0];
                ctcmin = LARGE;
                if((tl<-23.0) || ((tl<=-15.0)&&(tl>-16.0)) ||((tl<=-5.8) && (tl>-6.0)))
                {
                        tn=tl;
                        dix1=0;
                        dix2=0;
                        dix3=0;
                        price=LP;
                        tnhour = tn+24;
                        time = tn+24.0;
                        tz1s=TZ1SHI;
                        tz2s=TZ2SHI;
                        r1=R1;
                        r2=R2;
                        r3=R3;
                        a1=A1;
```

```
                         a2=A2;
                         qz1
=(qz1tab[tnhour]+qz2tab[tnhour]+qz3tab[tnhour]+qz4tab[tnhour])*3600;
                         qz2
=(qz5tab[tnhour]+qz6tab[tnhour]+qz7tab[tnhour]+qz8tab[tnhour])*3600;
                         pqz1=qz1tab[tnhour]*3600;
                            pqz2=qz2tab[tnhour]*3600;
                            pqz3=qz3tab[tnhour]*3600;
                            pqz4=qz4tab[tnhour]*3600;
                            pqz5=qz5tab[tnhour]*3600;
                            pqz6=qz6tab[tnhour]*3600;
                            pqz7=qz7tab[tnhour]*3600;
                            pqz8=qz8tab[tnhour]*3600;
                         TE = tetab[tnhour];
                         DTMAX=24.0+tn;
                          if((tl<=-15.0)&&(tl>-16.0))
                         {
                         tz1s=TZ1SLO;
                         tz2s=TZ2SLO;
                          price=HP;
                          DTMAX=16.0+tl;
                          }
                          if((tl<=-5.8)&&(tl>-6.0))
                          {
                          price=HP;
                          DTMAX=6.0+tl;
                          }
                         dt=DTMAX;
                         test1(1);

                   }
                 else
                 {
                 for(dix3=-1; dix3<=1; dix3++)
                 {
                         for(dix2=-1; dix2<=1; dix2++)
                         {
                                 for(dix1=-1; dix1<=1; dix1++)
                                 {
                                         tn = tltab[ix1n][ix2n][ix3n][0];
                                         tnhour = tn+24;
                                         time = tn+24.0;
                                           qz1
=(qz1tab[tnhour]+qz2tab[tnhour]+qz3tab[tnhour]+qz4tab[tnhour])*3600;
                                           qz2
=(qz5tab[tnhour]+qz6tab[tnhour]+qz7tab[tnhour]+qz8tab[tnhour])*3600;
                                         pqz1=qz1tab[tnhour]*3600;
                                         pqz2=qz2tab[tnhour]*3600;
                                         pqz3=qz3tab[tnhour]*3600;
                                         pqz4=qz4tab[tnhour]*3600;
                                         pqz5=qz5tab[tnhour]*3600;
                                         pqz6=qz6tab[tnhour]*3600;
                                         pqz7=qz7tab[tnhour]*3600;
                                         pqz8=qz8tab[tnhour]*3600;
                                         TE = tetab[tnhour];
                                         tz1s=TZ1SHI;
                                         tz2s=TZ2SHI;
                                         r1=R1;
                                         r2=R2;
                                         r3=R3;
                                         a1=A1;
                                         a2=A2;
                                         if((time>8.0) && (time<=18.0))
                                         {
                                                 tz1s=TZ1SLO;
                                                 tz2s=TZ2SLO;
                                         }
                                         if(time<=8.0)
                                         price=LP;
                                         else
                                         price=HP;
                                          if((tl<=-5.0)&&(tl>-6.0))
                                         {
                                         DTMAX=0.2;
                                         for(l=1; l>=1; l--)
```

```
                                    {
                                           dt=DTMAX/l;
                                           if((time>=tnhour) && (tnhour>(time-dt)))
                                           {
                                                 tnhour1=24+tn-dt;

qz1=((qz1tab[tnhour1]+qz2tab[tnhour1]+qz3tab[tnhour1]+qz4tab[tnhour1])*(tnhour-(time-
dt))/dt+(qz1tab[tnhour]+qz2tab[tnhour]+qz3tab[tnhour]+qz4tab[tnhour])
                                                 *(time-tnhour)/dt)*3600;

qz2=((qz5tab[tnhour1]+qz6tab[tnhour1]+qz7tab[tnhour1]+qz8tab[tnhour1])*(tnhour-(time-
dt))/dt+(qz5tab[tnhour]+qz6tab[tnhour]+qz7tab[tnhour]+qz8tab[tnhour])
                                                 *(time-tnhour)/dt)*3600;
                                                      pqz1=(qz1tab[tnhour1]*(tnhour
-(time-dt))/dt+qz1tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz2=(qz2tab[tnhour1]*(tnhour
-(time-dt))/dt+qz2tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz3=(qz3tab[tnhour1]*(tnhour
-(time-dt))/dt+qz3tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz4=(qz4tab[tnhour1]*(tnhour
-(time-dt))/dt+qz4tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz5=(qz5tab[tnhour1]*(tnhour
-(time-dt))/dt+qz5tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz6=(qz6tab[tnhour1]*(tnhour

-(time-dt))/dt+qz6tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz7=(qz7tab[tnhour1]*(tnhour
-(time-dt))/dt+qz7tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz8=(qz8tab[tnhour1]*(tnhour
-(time-dt))/dt+qz8tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;



                                                 TE=(tetab[tnhour1]*(tnhour-(time-
dt))/dt+tetab[tnhour]*(time-tnhour)/dt);

                                                 }
                                           test1(l);
                                           }
                                    }
                                    else
                                    {
                                    DTMAX=DTMAX1;

                                    for(l=LMAX; l>=1; l--)
                                    {
                                           dt=DTMAX/l;
                                           if((time>=tnhour) && (tnhour>(time-dt)))
                                           {
                                                 tnhour1=24+tn-dt;

qz1=((qz1tab[tnhour1]+qz2tab[tnhour1]+qz3tab[tnhour1]+qz4tab[tnhour1])*(tnhour-(time-
dt))/dt+(qz1tab[tnhour]+qz2tab[tnhour]+qz3tab[tnhour]+qz4tab[tnhour])
                                                 *(time-tnhour)/dt)*3600;

qz2=((qz5tab[tnhour1]+qz6tab[tnhour1]+qz7tab[tnhour1]+qz8tab[tnhour1])*(tnhour-(time-
dt))/dt+(qz5tab[tnhour]+qz6tab[tnhour]+qz7tab[tnhour]+qz8tab[tnhour])

                                                 *(time-tnhour)/dt)*3600;
                                                      pqz1=(qz1tab[tnhour1]*(tnhour
-(time-dt))/dt+qz1tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz2=(qz2tab[tnhour1]*(tnhour
-(time-dt))/dt+qz2tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz3=(qz3tab[tnhour1]*(tnhour
-(time-dt))/dt+qz3tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz4=(qz4tab[tnhour1]*(tnhour
-(time-dt))/dt+qz4tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz5=(qz5tab[tnhour1]*(tnhour
-(time-dt))/dt+qz5tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz6=(qz6tab[tnhour1]*(tnhour
-(time-dt))/dt+qz6tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz7=(qz7tab[tnhour1]*(tnhour
-(time-dt))/dt+qz7tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                      pqz8=(qz8tab[tnhour1]*(tnhour
-(time-dt))/dt+qz8tab[tnhour]
                                                      *(time-tnhour)/dt)*3600;
                                                 TE=(tetab[tnhour1]*(tnhour-(time-
dt))/dt+tetab[tnhour]*(time-tnhour)/dt);
                                                }
                                           test1(1);
                                    }
                                    }
                             }
                      }
               }
               }
               if(tlu >= tlmax)
               {
                      printf("\failed update");
                      break;
               }
               for(n=LENGTH-2; n>=0; n--)
               {
                      u1tab[ix1u][ix2u][ix3u][n+1] =
                             u1tab[ix1next][ix2next][ix3next][n];
                      u2tab[ix1u][ix2u][ix3u][n+1] =
                             u2tab[ix1next][ix2next][ix3next][n];
                      u3tab[ix1u][ix2u][ix3u][n+1] =
                             u3tab[ix1next][ix2next][ix3next][n];
                      tltab[ix1u][ix2u][ix3u][n+1] =
                             tltab[ix1next][ix2next][ix3next][n];
                      labtab[ix1u][ix2u][ix3u][n+1][0] =
                        labtab[ix1next][ix2next][ix3next][n][0];
                      labtab[ix1u][ix2u][ix3u][n+1][1] =
                        labtab[ix1next][ix2next][ix3next][n][1];
                      labtab[ix1u][ix2u][ix3u][n+1][2] =
                        labtab[ix1next][ix2next][ix3next][n][2];
               }
               cotab[ix1u][ix2u][ix3u] = nco;
               if (tlu<-16.0)
               {
                       ectab[ix1u][ix2u][ix3u][0] =
nec+ectab[ix1next][ix2next][ix3next][0];
                       ectab[ix1u][ix2u][ix3u][1] =
ectab[ix1next][ix2next][ix3next][1];
               }
               else
               {
                       ectab[ix1u][ix2u][ix3u][1] =
nec+ectab[ix1next][ix2next][ix3next][1];
                       ectab[ix1u][ix2u][ix3u][0] =
ectab[ix1next][ix2next][ix3next][0];
               }
                       chtab[ix1u][ix2u][ix3u] =
nchill+chtab[ix1next][ix2next][ix3next];

               if(tlu==-6.0||tlu==-16.0)
               {
               tltab[ix1u][ix2u][ix3u][0] = tlu-(rand()/32768.0)/1000000;
               }
               else
               {
               tltab[ix1u][ix2u][ix3u][0] = tlu;
```

```
                }
                u1tab[ix1u][ix2u][ix3u][0] = uu1;
                u2tab[ix1u][ix2u][ix3u][0] = uu2;
                u3tab[ix1u][ix2u][ix3u][0] = uu3;
                labtab[ix1u][ix2u][ix3u][0][0] = ix1u;
                labtab[ix1u][ix2u][ix3u][0][1] = ix2u;
                labtab[ix1u][ix2u][ix3u][0][2] = ix3u;
        } /* end of while loop */

        printf("\nLABLE TRACES");
        for(ix3=IX3MAX; ix3>=0; ix3--)
        {
                for(ix2=IX2MAX; ix2>=0; ix2--)
                {
                        for(ix1=0; ix1<=IX1MAX; ix1++)
                        {
                                for(n=0; n<LENGTH; n++)
                                {
                                        if(t1tab[ix1][ix2][ix3][n] <= -24.0)
                                        {first1=labtab[ix1][ix2][ix3][n][0];
                                         last1=labtab[ix1][ix2][ix3][LM1][0];
                                         first2=labtab[ix1][ix2][ix3][n][1];
                                         last2=labtab[ix1][ix2][ix3][LM1][1];
                                         first3=labtab[ix1][ix2][ix3][n][2];
                                         last3=labtab[ix1][ix2][ix3][LM1][2];
                                         if(first1==last1 && first2==last2 &&
first3==last3)
                                         {
                                         printf("\n");
                                                for(m=0; m<LENGTH; m++)
                                                {

        T[1][m]=(X1MIN+labtab[ix1][ix2][ix3][m][0]*DX1)/RCZ1;

        T[2][m]=(X2MIN+labtab[ix1][ix2][ix3][m][1]*DX2)/RCZ2;

        T[3][m]=(X3MIN+labtab[ix1][ix2][ix3][m][2]*DX3)/RCC;
                                                        COP[m]=(COPMAX-1)*(1-(T0-
T[3][m])/TMAX);

                                                        printf("\nlabeltrace[%2d] =
%2d,%2d,%2d", m,

                                                        labtab[ix1][ix2][ix3][m][0],
                                                        labtab[ix1][ix2][ix3][m][1],
                                                        labtab[ix1][ix2][ix3][m][2]);
                                                        printf("\tt1trace[%2d] = %6f", m,
                                                        t1tab[ix1][ix2][ix3][m]);

if(t1tab[ix1][ix2][ix3][m]>=0.0)
                                                               break;
                                                }
                                                      count=m;
                                                printf("\npeak-hour energy cost = %12f",
                                                ectab[first1][first2][first3][1]);
                                                printf("\noff peak-hour energy cost =
%12f",

                                                ectab[first1][first2][first3][0]);
                                                printf("\nchiller energy cost = %12f",
                                                chtab[first1][first2][first3]);
                                             printf("\n");
                                        plot();
                                        }/* end of if t1tab <= -24.0 */
                                         }/* end of uf first1== last1 etc.*/
                                }
                        }
                }
        }
}
/****** end of main() *****/


/*********functions to compute new-node index*************/

int fix1n()
{
        extern int ix1u, dix1;
        int i;
        if((ix1u==0 && dix1<0)||(ix1u==IX1MAX && dix1>0)) i=0;
        else i=dix1;
        return(ix1u+i);
}

int fix2n()
{
        extern int ix2u, dix2;
        int j;
        if((ix2u==0 && dix2<0)||(ix2u==IX2MAX && dix2>0)) j=0;
        else j=dix2;
        return(ix2u+j);
}

int fix3n()
{
        extern int ix3u, dix3;
        int k;
        if((ix3u==0 && dix3<0)||(ix3u==IX3MAX && dix3>0)) k=0;
        else k=dix3;
        return(ix3u+k);
}


/************* procedure to call trial cost computation *****/

void test1(int l)
{
        v1 = (ix1n-ix1u)*DX1/dt - d1 + x1u*AZ1/CZ1;
        v2 = (ix2n-ix2u)*DX2/dt - d2 + x2u*AZ2/CZ2;
        v3 = (ix3n-ix3u)*DX3/dt - d3 + x3u*AC/CC;
        u1 = v1/(B1*g1);
        u2 = v2/(B2*g2);
        u3 = (RG1*v1 + RG2*v2 + v3)/(C33 + B3*g3);
        if((u1>=0.0) && (u2>=0.0) && (u3>=0.0))
        {
                test();
        }
        return;
}


/*****  procedure to compute trial cost *****/

void test()
{
        double tc, ct, ctc,tltry;
        tltry=tn-dt;
        if((tltry<-6.0)&&(tl>-5.0))return;
        if((tltry<-16.0)&&(tl>-15.0))return;
        if((tltry<-24.0)&&(tl>-23.0))return;
         pu1=u1*pqz1/qz1, pu2=u1*pqz2/qz1;
         pu3=u1*pqz3/qz1, pu4=u1*pqz4/qz1;
         pu5=u2*pqz5/qz2; pu6=u2*pqz6/qz2;
         pu7=u2*pqz7/qz2; pu8=u2*pqz8/qz2;
        tc = (( r1*pu1*pu1 + r1*pu2*pu2 + r1*pu3*pu3 +r1*pu4*pu4 +
                r2*pu5*pu5 + r2*pu6*pu6 + r2*pu7*pu7 +r2*pu8*pu8 +
                r3*u3 )*price+ a1*pow(x1u-X1S,2.0)/CZ1 +
                   a2*pow(x2u-X2S,2.0)/CZ2)*dt
                 + cotab[ix1n][ix2n][ix3n];
        ct = (tc - cotab[ix1u][ix2u][ix3u]) * (tn-dt-tm)/(tl-(tn-dt));
        ctc = tc + ct;
        if(ctc<=ctcmin)
        {
                ctcmin=ctc;
                nco=tc;
                tlu=tn-dt;
                uu1=u1;
                uu2=u2;
                uu3=u3;
                nec = ( r1*pu1*pu1 + r1*pu2*pu2 + r1*pu3*pu3 +r1*pu4*pu4 +
              r2*pu5*pu5 + r2*pu6*pu6 + r2*pu7*pu7 +r2*pu8*pu8 +
              r3*u3 )*dt;
              nchill=r3*u3*dt;
              ix1next = ix1n;
              ix2next = ix2n;
              ix3next = ix3n;
       }
       return;
}


/***************output time and u1 in plotU1 file*****************/
void plot()
{
        int n;
        FILE *fp, *fopen();
       fp=fopen("input1","w");
        fprintf(fp,"%2d",count);
       for(n=0; n<=count; n++)
       {
               m=count-n;
              fprintf(fp,"\n%6f",T[1][m]);
              fprintf(fp,"     %6f",T[2][m]);
              fprintf(fp,"     %6f",T[3][m]);
              fprintf(fp,"\n%6f",u1tab[ix1][ix2][ix3][m]);
              fprintf(fp,"     %6f",u2tab[ix1][ix2][ix3][m]);
              fprintf(fp,"     %6f",(t1tab[ix1][ix2][ix3][m]+24.0));
        }
       fclose(fp);
       return;
}
```

```
/***********************************************************************************
********
*
*      File Name:     dagg8.h
*      Purpose:       This is header file for eight-zone disaggregation. It  declares
*                     variables and functions required by the file dagg8.c, dinp11.c,
*                     dinp12.c, dinp21.c dinp22.c dinp23.c dinp24.c.
*
************************************************************************************
******/

#include <stdio.h>
#include <math.h>

#define LENGTH     (int)100                                        /*max length of
trajectories*/
#define LM1        (int)(LENGTH-1)
#define LARGE      (double)1000000000.0
#define LP         (double).5
#define HP         (double)1.0
#define TZ1SLO     (double)24.0                                    /*setpoint temp for
occupied zone*/
#define TZ1SHI     (double)30.0                                    /*setpoint temp for
unoccupied zone*/
#define TZ2SLO     (double)24.0
#define TZ2SHI     (double)30.0
#define TZ3SLO     (double)24.0
#define TZ3SHI     (double)30.0
#define TZ4SLO     (double)24.0
#define TZ4SHI     (double)30.0
#define IX1SPAN    (int)9
#define IX1MAX     (int)(IX1SPAN-1)
#define TZ1MIN     (double)(TZ1SLO-1.0)                            /*max zone temperature
constraint*/
```

```
#define TZ1MAX      (double)(TZ1SHI+1.0)                    /*max zone temperature
constraint*/
#define IX2SPAN     (int)9
#define IX2MAX      (int)(IX2SPAN-1)
#define TZ2MIN      (double)(TZ2SLO-1.0)
#define TZ2MAX      (double)(TZ2SHI+1.0)
#define A1          (double)12000.0
#define A2          (double)12000.0
#define R1          (double)0.00003
#define R2          (double)0.00003
#define CC          (double)250000.
#define G2          (double)CZ2/CC
#define G1          (double)CZ1/CC
#define ACZ         (double)0.75                            /*heat exchanger coef*/
#define B1          (double)(-ACZ/CZ1)
#define B2          (double)(-ACZ/CZ2)
#define DTMAX1      (double)1.0                             /*max time-step allowed*/
#define LMAX        (int)5                               /*number of control amplitudes
tested*/
#define X1MIN       (double)(RCZ1*TZ1MIN)                   /*min value, normalized
state variable*/
#define X1MAX       (double)(RCZ1*TZ1MAX)                   /*max value, normalized
state variable*/
#define DX1         (double)((X1MAX-X1MIN)/IX1MAX)          /*stepsize, normalized
state var*/
#define X2MIN       (double)(RCZ2*TZ2MIN)
#define X2MAX       (double)(RCZ2*TZ2MAX)
#define DX2         (double)((X2MAX-X2MIN)/IX2MAX)
#define X1S         (double)(RCZ1*tz1s)                     /*desired value, normalized
state var*/
#define X2S         (double)(RCZ2*tz2s)
#define x1u         (double)(X1MIN + ix1u*DX1)              /*discrete level, node
being updated*/
#define x2u         (double)(X2MIN + ix2u*DX2)
#define g1          (double)(x1u-RG1*CT*RCC)
#define g2          (double)(x2u-RG2*CT*RCC)
#define d1          (double)((qz1+AZ1*35.0)/RCZ1)             /*normalized zone cooling
load*/
#define d2          (double)((qz2+AZ2*35.0)/RCZ2)
#define ix1n        (int)fix1n()                            /*index of next node
examined*/
#define ix2n        (int)fix2n()
#define x1n         (double)(X1MIN + ix1n*DX1)              /*discrete level, next node
examined*/
#define x2n         (double)(X2MIN + ix2n*DX2)


/*********** function declarations ************/
void main(void);
int  fix1n(void);
int  fix2n(void);
void test(void);
void test1(int);
void plot(void);
void ice(void);
void dgata(void);

/*********** global variables *****************/
double  cotab[IX1SPAN][IX2SPAN];
double  lqatab[IX1SPAN][IX2SPAN][LENGTH];
double  t1tab[IX1SPAN][IX2SPAN][LENGTH];                 /*table of update times*/
double  ectab[IX1SPAN][IX2SPAN][2];                      /*table of update times*/
double  TCtab[IX1SPAN][IX2SPAN][LENGTH];                 /*table of u1*/
double  u1tab[IX1SPAN][IX2SPAN][LENGTH];                 /*table of u1*/
double  u2tab[IX1SPAN][IX2SPAN][LENGTH];                 /*table of u2*/
int     labtab[IX1SPAN][IX2SPAN][LENGTH][2];              /*table of lable-strings*/
int     ix1u,ix2u,ix1,ix2,m,count, tnhour, numb;
int     dix1,dix2,ix1next,ix2next;
double  sqz1,sqz2,spqz1,spqz2,spqz3,spqz4,sprice,sqa,stz1s,stz2s,sla;
double  u1,u2,v1,v2,RG1,RG2,RCZ1,RCZ2,qz1,RCC,qz2,qz3,qz4,tz1s,tz2s,DTMAX;
double
tn,tm,dt,t1,uu1,uu2,t1u,nco,ctcmin,a1,a2,nec,CT,CCT,TE,tme,q,qa,nlq,price,la,tqa;
double
TC[200],TZag1[200],TZag2[200],TZ[200],Uag1[200],Uag2[200],U[200],T[200],lqa[200];
```

```
/*****************************************************************************************
********
*
*       File Name:      daggl1.h
*       Purpose:        This is header file for eight-zone disaggregation. It
declares
*                       variables used to disaggregate the first microzone.
*
*****************************************************************************************
******/

#include <stdio.h>
#include <math.h>

#defineCZR1     (double)(374.0)                                 /*zone heat capacity, kw
hours/deg C*/
#defineCZR2     (double)(370.0)
#defineCZR3     (double)(300.0)
#defineCZR4     (double)(187.0)
#defineCZ1      (double)(CZR1+CZR2)
#defineCZ2      (double)(CZR3+CZR4)
#defineAZR1     (double)(410.0)
#defineAZR2     (double)(400.0)
#defineAZR3     (double)(330.0)
#defineAZR4     (double)(205.0)
#defineAZ1      (double)(AZR1+AZR2)                             /*heat transfer coef for
zone*/
#defineAZ2      (double)(AZR3+AZR4)

double  qz1tab[25] ={3.682+3.682,2.987+2.978,2.275+2.275,1.517+1.517,0.517+
                 0.517,0.517+0.517,2.275+2.275,4.034+4.034,5.792+5.792,
                7.550+7.550,9.309+9.309,11.067+11.067,12.826+12.826,14.584+
                14.584,16.343+16.343,18.101+18.101,16.343+16.343,
                 14.584+14.584,11.067+11.067,7.550+7.550,5.792+5.792,5.088+
                5.088,4.385+4.385,3.682+3.682,3.682+3.682};
/*lists of hourly zone cooling loads*/
double qz2tab[25] = {2.987+1.814,2.275+1.489,1.572+1.127,0.517+0.758,0.517+
                0.208,0.517+0.208,2.275+1.237,3.330+2.017,4.034+2.846,
                5.792+3.770,6.847+4.604,7.550+5.533,8.254+6.413,9.309+7.294,
                11.067+8.171,10.364+9.050,9.661+8.171,
                 8.957+7.292,8.254+5.533,7.550+3.770,5.792+2.891,5.088+2.544,
                4.385+2.192,3.682+1.841,2.987+1.841};
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                35.0,35.0};
/*lists of hourly envioment temperture*/
double  pqz1tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
                 7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
                 14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;
double  pqz2tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
                 7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
                 14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;
double  pqz3tab[25] = {2.987,2.275,1.572,0.517,0.517,0.517,2.275,3.330,4.034,
                 5.792,6.847,7.550,8.254,9.309,11.067,10.364,9.661,
                 8.957,8.254,7.550,5.792,5.088,4.385,3.682,2.987} ;
double  pqz4tab[25] = {1.841,1.489,1.127,0.758,0.208,0.208,1.237,2.017,2.846,
                 3.770,4.604,05.533,06.413,07.294,08.171,09.050,08.171,
                 07.292,05.533,3.770,2.891,2.544,2.192,1.841,1.841};

double pu1, pu2, pu3, pu4, pqz1, pqz2, pqz3, pqz4;
double pp = 0;
char outfile[] = "input21";
```

```
/*******************************************************************************************************
********
*
*       File Name:      dagg12.h
*       Purpose:        This is header file for eight-zone disaggregation. It
declares
*                       variables used to disaggregate the second microzone.
*
********************************************************************************************************
******/

#include <stdio.h>
#include <math.h>

#defineCZR5    (double)(374.0)
#defineCZR6    (double)(300.0)
#defineCZR7    (double)(187.0)
#defineCZR8    (double)(100.0)
#defineCZ1     (double)(CZR5+CZR6)
#defineCZ2     (double)(CZR7+CZR8)
#defineAZR5    (double)(410.0)
#defineAZR6    (double)(330.0)
#defineAZR7    (double)(205.0)
#defineAZR8    (double)(110.0)
#defineAZ1     (double)(AZR5+AZR6)                          /*heat transfer coef for
zone*/
#defineAZ2     (double)(AZR7+AZR8)

double qz1tab[25] = {8.101+8.101,8.101+8.101,8.101+8.101,8.101+8.101,8.101+
                 8.101,8.101+8.101,8.101+8.101,8.101+8.101,8.101+8.101,
                 8.101+8.101,8.101+8.101,8.101+8.101,8.101+8.101,8.101+8.101,
                 8.101+8.101,8.101+8.101,8.101+8.101,
                   8.101+8.101,8.101+8.101,8.101+8.101,8.101+8.101,8.101+8.101,
                 8.101+8.101,8.101+8.101,8.101+8.101} ;
double  qz2tab[25] = {5.223+3.223,5.223+3.223,5.223+3.223,5.223+3.223,5.223+
                 3.223,5.223+3.223,5.223+3.223,5.223+3.223,5.223+3.223,
                 5.223+3.223,5.223+3.223,5.223+3.223,5.223+3.223,5.223+3.223,
                 5.223+3.223,5.223+3.223,5.223+3.223,
                 5.223+3.223,5.223+3.223,5.223+3.223,5.223+3.223,5.223+3.223,
                 3.223+3.223,5.223+3.223,5.223+3.223};
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                 35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                 35.0,35.0};
double  pqz1tab[25] = {8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                   8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                   8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101} ;
double  pqz2tab[25] = {8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                   8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                   8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101} ;
double  pqz3tab[25] = {5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
                   5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
                   5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223};
double  pqz4tab[25] = {3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,
                   3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,
                   3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223};

double pu1, pu2, pu3, pu4, pqz1, pqz2, pqz3, pqz4;
double pp=0.0;
char   outfile[] = "input22";




/*******************************************************************************************************
*********
*
*       File Name:      dagg4.c
*                     This program computes the optimal trajectories of the
*                     twi smaller microzonez.
*                        The main function calls module:
```

```
*                               fix1n()
*                               fix2n()
*                               test()
*                               test1(int)
*                               plot()
*                           icd()
*                           gdata()
*
************************************************************************************
*******/

#include <math.h>
#include "dagg.h"
#include "dagg8.h"

void main()
{
        double  tlmax=0.0;                      /*start at midnight, go backwards
(neg time)*/
        double  tmn;
        int     tnhour1;
        int     l, n, first1, first2, last1, last2 ;


        /**** initialize the arrays *****/

        RG1 = sqrt(G1);
        RG2 = sqrt(G2);
        RCZ1 = sqrt(CZ1);
        RCZ2 = sqrt(CZ2);
        RCC = sqrt(CC);
        for(ix1=0; ix1<=IX1MAX; ix1++)
        {       for(ix2=0; ix2<=IX2MAX; ix2++)
                {
                     cotab[ix1][ix2]=0.0;
                        ectab[ix1][ix2][0]=0.0;
                        ectab[ix1][ix2][1]=0.0;
                        for(m=0; m<LENGTH; m++)
                        {
                            tltab[ix1][ix2][m] = 0.0+(rand()/32768.0)/1000000;
                                lqatab[ix1][ix2][m]=0.1;
                                TCtab[ix1][ix2][m] = 0.0;
                                u1tab[ix1][ix2][m] = 0.0;
                                u2tab[ix1][ix2][m] = 0.0;
                                labtab[ix1][ix2][m][0] = ix1;      /*give each node
its own lable*/
                                labtab[ix1][ix2][m][1] = ix2;
                     }
                }
       }

       /**** read data for data file ****/

       gdata();
        while( tlmax > -24.0 )
       {

       /**** find the next node to update ****/

        tlmax = -24.0;
        for(ix1=0; ix1<=IX1MAX; ix1++)
        {
                for(ix2=0; ix?<=IX2MAX; ix2++)
                {
                     if(tltab[ix1][ix2][0]>tlmax)
                        {
                            ix1u=ix1;           /*node to be updated has latest last
update*/
                                ix2u=ix2;
                                tlmax=tltab[ix1][ix2][0];
                                tlu = tlmax;
                        }
                }
        }
```

```
/**** find the earliest of the admissible update times*****/

 tm = 0.0;
 for(dix2=-1; dix2<=1; dix2++)
 {
       for(dix1=-1; dix1<=1; dix1++)
       {
              tn=t1tab[ix1n][ix2n][0];
                 tmn = tn-DTMAX1;
              if(tmn<=tm) tm=tmn;
         }
 }

/*** find the least-cost transition ***/

         t1 = t1tab[ix1u][ix2u][0];
         ctcmin = LARGE;
         if((t1<-23.0)||((t1<=-15.0)&&(t1>-16.0)) ||((t1<=-5.8)&&(t1>-6.0)))
         {
                tn=t1;
                dix1=0;
                dix2=0;
              price=LP;
                tnhour = tn+24;
                tme = (tn+24.0);
                tz1s=TZ1SHI;
                tz2s=TZ2SHI;
                qz1 = qz1tab[tnhour]*3600;
                qz2 = qz2tab[tnhour]*3600;
                pqz1=pqz1tab[tnhour]*3600;
                pqz2=pqz2tab[tnhour]*3600;
                pqz3=pqz3tab[tnhour]*3600;
                pqz4=pqz4tab[tnhour]*3600;
                TE = tetab[tnhour];
                DTMAX=24.0+tn;
                if((t1<=-15.0)&&(t1>-16.0))
                {
                tz1s=TZ1SLO;
                tz2s=TZ2SLO;
              price=HP;
                DTMAX=16.0+t1;
                }
                if((t1<=-5.8)&&(t1>-6.0))
                {
              price=HP;
                DTMAX=6.0+t1;
                }
                 dt=DTMAX;
                 ice();
                 test1(1);

         }
         else
         {
                 for(dix2=-1; dix2<=1; dix2++)
                 {
                         for(dix1=-1; dix1<=1; dix1++)
                         {
                                 tn = t1tab[ix1n][ix2n][0];
                                 tnhour = tn+24;
                                 tme =tn+24.0;
                                 qz1 = qz1tab[tnhour]*3600;
                                 qz2 = qz2tab[tnhour]*3600;
                                    pqz1=pqz1tab[tnhour]*3600;
                                    pqz2=pqz2tab[tnhour]*3600;
                                    pqz3=pqz3tab[tnhour]*3600;
                                    pqz4=pqz4tab[tnhour]*3600;
                                 TE = tetab[tnhour];
                                 tz1s=TZ1SHI;
                                 tz2s=TZ2SHI;
                                 if((tme>8.0) && (tme<=18.0))
                                 {
                                         tz1s=TZ1SLO;
                                         tz2s=TZ2SLO;
                                 }
```

```
                                    if(tme<=8.0)
                                    price=LP;
                                    else
                                    price=HP;
                                    if((tl<=-5.0) &&(tl>-6.0))
                                    {
                                    DTMAX=0.2;
                                        for(l=1; l>=1; l--)
                                        {
                                                dt=DTMAX/l;
                                                if((tme>=tnhour) && (tnhour>(tme-
dt)))
                                                {
                                                      tnhour1=24+tn-dt;
                                                      qz1=(qz1tab[tnhour1]*(tnhour-
(tme-dt))/dt+qz1tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      qz2=(qz2tab[tnhour1]*(tnhour-
(tme-dt))/dt+qz2tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      pqz1=(pqz1tab[tnhour1]*(tnhour
-(tme-dt))/dt+pqz1tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      pqz2=(pqz2tab[tnhour1]*(tnhour
-(tme-dt))/dt+pqz2tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      pqz3=(pqz3tab[tnhour1]*(tnhour
-(tme-dt))/dt+pqz3tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      pqz4=(pqz4tab[tnhour1]*(tnhour
-(tme-dt))/dt+pqz4tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      TE=(tetab[tnhour1]*(tnhour-
(tme-dt))/dt+tetab[tnhour]*(tme-tnhour)/dt);

                                                }
                                                ice();
                                           if((dix1==0) && (dix2==0) &&(dt==DTMAX/l))
                                           {
                                           sqa=qa;
                                           sqz1=qz1;
                                           sqz2=qz2;
                                           spqz1 = pqz1;
                                           spqz2 = pqz2;
                                           spqz3 = pqz3;
                                           spqz4 = pqz4;
                                           stz1s=tz1s;
                                           stz2s=tz2s;
                                           sla=la;
                                           sprice=price;
                                           }
                                                test1(l);
                                           }
                                        }
                                    else
                                    {
                                        DTMAX=DTMAX1;

                                        for(l=LMAX; l>=1; l--)
                                        {
                                                dt=DTMAX/l;
                                                if((tme>=tnhour) && (tnhour>(tme-
dt)))
                                                {
                                                      tnhour1=24+tn-dt;
                                                      qz1=(qz1tab[tnhour1]*(tnhour-
(tme-dt))/dt+qz1tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      qz2=(qz2tab[tnhour1]*(tnhour-
(tme-dt))/dt+qz2tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      pqz1=(pqz1tab[tnhour1]*(tnhour
-(tme-dt))/dt+pqz1tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      pqz2=(pqz2tab[tnhour1]*(tnhour
-(tme-dt))/dt+pqz2tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      pqz3=(pqz3tab[tnhour1]*(tnhour

-(tme-dt))/dt+pqz3tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;
                                                      pqz4=(pqz4tab[tnhour1]*(tnhour

-(tme-dt))/dt+pqz4tab[tnhour]
                                                      *(tme-tnhour)/dt)*3600;

                                                      TE=(tetab[tnhour1]*(tnhour-

(tme-dt))/dt+tetab[tnhour]*(tme-tnhour)/dt);

                                                 }
                                                 ice();
                                            if((dix1==0) && (dix2==0)

&&(dt==DTMAX/5)){

                                            sqa=qa;
                                            sqz1=qz1;
                                            sqz2=qz2;
                                            spqz1 = pqz1;
                                            spqz2 = pqz2;
                                            spqz3 = pqz3;
                                            spqz4 = pqz4;
                                            stz1s=tz1s;
                                            stz2s=tz2s;
                                            sla=la;
                                            sprice=price;
                                            }
                                                 test1(1);
                                    }/* end of for */
                                        }/* end of else */
                                 }
                         }
                 }

                 if(tlu >= tlmax)
                 {
                         printf("\failed update");
                         break;
                 }
                 for(n=LENGTH-2; n>=0; n--)
                 {
                         TCtab[ix1u][ix2u][n+1] =
                                 TCtab[ix1next][ix2next][n];
                         u1tab[ix1u][ix2u][n+1] =
                                 u1tab[ix1next][ix2next][n];
                         u2tab[ix1u][ix2u][n+1] =
                                 u2tab[ix1next][ix2next][n];
                         t1tab[ix1u][ix2u][n+1] =
                                 t1tab[ix1next][ix2next][n];
                         lqatab[ix1u][ix2u][n+1] =
                                 lqatab[ix1next][ix2next][n];
                         labtab[ix1u][ix2u][n+1][0] =
                           labtab[ix1next][ix2next][n][0];
                         labtab[ix1u][ix2u][n+1][1] =
                           labtab[ix1next][ix2next][n][1];
                 }
                 cotab[ix1u][ix2u] = nco;
               if (tlu<-16.0)
               {
               ectab[ix1u][ix2u][0]=nec+ectab[ix1next][ix2next][0];
               ectab[ix1u][ix2u][1]=ectab[ix1next][ix2next][1];
               }
               else
               {
               ectab[ix1u][ix2u][1]=nec+ectab[ix1next][ix2next][1];
               ectab[ix1u][ix2u][0]=ectab[ix1next][ix2next][0];
               }
                 lqatab[ix1u][ix2u][0] = nlq;
                 if(tlu==-6.0||tlu==-16.0)
                 {
                 tltab[ix1u][ix2u][0] = tlu-(rand()/32768.0)/1000000;
                 }
                 else
                 {
```

```
                tltab[ix1u][ix2u][0] = tlu;
                }
                ultab[ix1u][ix2u][0] = uu1;
                u2tab[ix1u][ix2u][0] = uu2;
                TCtab[ix1u][ix2u][0] = CCT;
                labtab[ix1u][ix2u][0][0] = ix1u;
                labtab[ix1u][ix2u][0][1] = ix2u;
     } /*** closes the "while" loop in main() ***/
        printf("\nlabel trace");

                for(ix2=IX2MAX; ix2>=0; ix2--)
                {
                        for(ix1=0; ix1<=IX1MAX; ix1++)
                        {
                                for(n=0; n<LENGTH; n++)
                                {
                                        if(tltab[ix1][ix2][n] <= -24.0)
                                        {first1=labtab[ix1][ix2][n][0];
                                         last1=labtab[ix1][ix2][LM1][0];
                                         first2=labtab[ix1][ix2][n][1];
                                         last2=labtab[ix1][ix2][LM1][1];
                                         if(first1==last1 &&first2==last2 )
                                         {
                                         printf("\n");
                                                for(m=0; m<LENGTH; m++)
                                                {
                                                        printf("\nlabeltrace[%2d] =
%2d,%2d", m,
                                                        labtab[ix1][ix2][m][0],
                                                        labtab[ix1][ix2][m][1]);
                                                        printf("     tltrace[%2d] =
%6f", m,
                                                        tltab[ix1][ix2][m]+24.0);
                                                        printf("\nultrace[%2d] =
%6f", m,
                                                        ultab[ix1][ix2][m]);
                                                        printf("     u2trace[%2d] =
%6f", m,
                                                        u2tab[ix1][ix2][m]);

printf("\nlqatab=%12f",lqatab[ix1][ix2][m]);
                                                  if(tltab[ix1][ix2][m]>=0.0) break;
                                                }
                                          numb=m;

       printf("\npeak=%12f",ectab[first1][first2][1]);
                                          printf("\noff
peak=%12f",ectab[first1][first2][0]);
                                          plot();
                                        }/* end of if */
                                       }/* end of if */
                            }/* end of for */
                     }/* end of for */
              }/* end of for*/
}
/****** end of main() *****/


/*********functions to compute new-node index*************/

 int fix1n()
 {
        extern int ix1u, dix1;
       int i;
        if((ix1u<=1 && dix1==-2)||(ix1u==0 && dix1==-1)||(ix1u==IX1MAX &&
dix1==1)||(ix1u>=(IX1MAX-1) && dix1==2)) i=0;
        else i=dix1;
        return(ix1u+i);
 }

int fix2n()
{
        extern int ix2u, dix2;
        int j;
        if((ix2u==0 && dix2==-1)||(ix2u<=1 && dix2==-2)||(ix2u==IX2MAX && dix2==1)
||(ix2u>=(IX2MAX-1) && dix2==2)) j=0;
        else j=dix2;
        return(ix2u+j);
}
/************** procedure to call trial cost computation *****/

void test1(int l)
{
        v1 = (ix1n-ix1u)*DX1/dt - d1 + x1u*AZ1/CZ1;
        v2 = (ix2n-ix2u)*DX2/dt - d2 + x2u*AZ2/CZ2;
        u1 = v1/(B1*g1);
        u2 = v2/(B2*g2);
        q = (u1*g1/RCZ1 + u2*g2/RCZ2)*ACZ*dt;
        tqa = qa+lqatab[ix1n][ix2n][0]+la*pp;
        if((u1>=0.0) && (u2>=0.0) &&(q<=tqa) ){

                test();
        }
        if (((tl<-23.0)&&(tl==tlu)) ||((tl<=-15.)&&(tl>-16)&&(tl==tlu))||((tl<=-
5.8)&&(tl>-6.0)&&(tl==tlu))){
                test();
                return;
        }
        if ((((DTMAX==0.2)&&(dt==0.2))||((DTMAX==1.0)&&(dt==1.0))) && (tlu==tl) &&
(dix1==1) && (dix2==1)) {
        dix1=0;
       dix2=0;
       dt=0.2;
       tn=tltab[ix1n][ix2n][0];
       qz1=sqz1;
       qz2=sqz2;
       pqz1=spqz1;
       pqz2=spqz2;
       pqz3=spqz3;
       pqz4=spqz4;
       tz1s=stz1s;
       tz2s=stz2s;
       price=sprice;
       la=sla;
       qa=sqa;
       v1 = (ix1n-ix1u)*DX1/dt - d1 + x1u*AZ1/CZ1;
        v2 = (ix2n-ix2u)*DX2/dt - d2 + x2u*AZ2/CZ2;
        u1 = v1/(B1*g1);
        u2 = v2/(B2*g2);
        q = (u1*g1/RCZ1 + u2*g2/RCZ2)*ACZ*dt;
        tqa=qa+lqatab[ix1n][ix2n][0]+la*pp;
           test();
           dix1=1;
           dix2=1;
           return;
       }
        return;
}
/*****  procedure to compute trial cost *****/

void test()
{
        double tc, ct, ctc,tltry;
        tltry=tn-dt;
        if((tltry<-6.0)&&(tl>-5.8))return;
        if((tltry<-16.0)&&(tl>-15.0))return;
        if((tltry<-24.0)&&(tl>-23.0))return;
       if((tn>-6.0)&&(tn<-1.0)&&(((ix1u-ix1n)>0)||((ix2u-ix2n)>0)))return;
       pu1 = u1*pqz1/qz1; pu2 = u1*pqz2/qz1;
       pu3 = u2*pqz3/qz2; pu4 = u1*pqz4/qz2;
        tc = ((R1*pu1*pu1 + R1*pu2*pu2 + R2*pu3*pu3 + R2*pu4*pu4 )*price+ A1*pow(x1u-
X1S,2.0)/CZ1 +
                   A2*pow(x2u-X2S,2.0)/CZ2)*dt
               + cotab[ix1n][ix2n];
        ct = (tc - cotab[ix1u][ix2u]) * (tn-dt-tm)/(tl-(tn-dt));
        ctc = tc + ct;

        if(ctc<=ctcmin)
        {
                ctcmin=ctc;
                nco=tc;
                nlq=tqa-q;
                tlu=tn-dt;
                uu1=u1;
                uu2=u2;
              CCT=CT;
                nec=(R1*pu1*pu1 + R1*pu2*pu2 + R2*pu3*pu3 + R2*pu4*pu4)*dt;
                ix1next = ix1n;
                ix2next = ix2n;
        }
        return;
}

/***** compute possible qa and CT at the time period of dt ********/

void ice()
{
       int i;
             for(i=0; i<count; i++)
             {
                   T[0]=1.0+T[0];
                  if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+1])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*dt;
                         CT=TC[i+1];
                      la=(lqa[i+1]-lqa[i])*dt/(T[i]-T[i+1]);
                         break;
                   }
                  if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+2])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-tme+dt)/dt;
                         la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                          (lqa[i+2]-lqa[i+1])*(T[i+1]-tme+dt)/(T[i+1]-T[i+2]);
                      break;
                   }
                  if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+3])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-T[i+2])
                           +U[i+3]*ACZ*(TZ[i+3]-TC[i+3])*(T[i+2]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-
T[i+2])/dt+TC[i+3]*(T[i+2]-tme+dt)/dt;
                         la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                          (lqa[i+2]-lqa[i+1])+
                          (lqa[i+3]-lqa[i+2])*(T[i+2]-tme+dt)/(T[i+2]-T[i+3]);
                      break;
                  }
                    if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+4])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-T[i+2])
                           +U[i+3]*ACZ*(TZ[i+3]-TC[i+3])*(T[i+2]-T[i+3])
                           +U[i+4]*ACZ*(TZ[i+4]-TC[i+4])*(T[i+3]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-T[i+2])/dt
                           +TC[i+3]*(T[i+2]-T[i+3])/dt
                           +TC[i+4]*(T[i+3]-tme+dt)/dt;
                         la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                          (lqa[i+2]-lqa[i+1])+
                          (lqa[i+3]-lqa[i+2])+
                          (lqa[i+4]-lqa[i+3])*(T[i+3]-tme+dt)/(T[i+3]-T[i+4]);
                      break;
                  }
                   if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+5])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-T[i+2])
                           +U[i+3]*ACZ*(TZ[i+3]-TC[i+3])*(T[i+2]-T[i+3])
                           +U[i+4]*ACZ*(TZ[i+4]-TC[i+4])*(T[i+3]-T[i+4])
                           +U[i+5]*ACZ*(TZ[i+5]-TC[i+5])*(T[i+4]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-T[i+2])/dt
                           +TC[i+3]*(T[i+2]-T[i+3])/dt
                           +TC[i+4]*(T[i+3]-T[i+4])/dt
```

```
                            +TC[i+5]*(T[i+4]-tme+dt)/dt;
                          la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                           (lqa[i+2]-lqa[i+1])+
                           (lqa[i+3]-lqa[i+2])+
                           (lqa[i+4]-lqa[i+3])+
                           (lqa[i+5]-lqa[i+4])*(T[i+4]-tme+dt)/(T[i+4]-T[i+5]);
                          break;
                   }
                    if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+6])
                    {
                          qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                            +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-T[i+2])
                            +U[i+3]*ACZ*(TZ[i+3]-TC[i+3])*(T[i+2]-T[i+3])
                            +U[i+4]*ACZ*(TZ[i+4]-TC[i+4])*(T[i+3]-T[i+4])
                            +U[i+5]*ACZ*(TZ[i+5]-TC[i+5])*(T[i+4]-T[i+5])
                            +U[i+6]*ACZ*(TZ[i+6]-TC[i+6])*(T[i+5]-tme+dt);
                          CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-T[i+2])/dt
                            +TC[i+3]*(T[i+2]-T[i+3])/dt
                            +TC[i+4]*(T[i+3]-T[i+4])/dt
                            +TC[i+5]*(T[i+4]-T[i+5])/dt
                            +TC[i+6]*(T[i+5]-tme+dt)/dt;
                          la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                           (lqa[i+2]-lqa[i+1])+
                           (lqa[i+3]-lqa[i+2])+
                           (lqa[i+4]-lqa[i+3])+
                           (lqa[i+5]-lqa[i+4])+
                           (lqa[i+6]-lqa[i+5])*(T[i+5]-tme+dt)/(T[i+5]-T[i+6]);
                          break;
                   }
          }
}

void plot()
{
        int n;
        FILE *fp, *fopen();
        fp=fopen(outfile,"w");
        fprintf(fp,"%2d",numb);
        for(n=0; n<=numb; n++)
        [
            m=numb-n;
            fprintf(fp,"\n%6f",(X1MIN+labtab[ix1][ix2][m][0]*DX1)/RCZ1);
            fprintf(fp,"    %6f",(X2MIN+labtab[ix1][ix2][m][1]*DX2)/RCZ2);
            fprintf(fp,"    %6f",TCtab[ix1][ix2][m]);
            fprintf(fp,"\n%6f",u1tab[ix1][ix2][m]);
            fprintf(fp,"    %6f",u2tab[ix1][ix2][m]);
            fprintf(fp,"    %6f",(t1tab[ix1][ix2][m]+24.0));
            fprintf(fp,"\n%6f",lqatab[ix1][ix2][m]);
         }
        fclose(fp);
        return;

}




/*****************************************************************************************
*********
*
*       File Name:      dinpl1.c
*       Purpose:        This program reads temperature and control values for two
microzone,
*                       temperature for storage, and time form input1 file. This
module
*                    is called by main function to disaggregate the first microzone.
*
******************************************************************************************
*******/

#include <stdio.h>
```

```
#include "dagg8.h"
/***** get data from data-file *************/

 void gdata()
{
        int i;
        char putfile[] = "input1";
        FILE *fp;
        fp=fopen(putfile,"r");
        fscanf(fp,"%d",&count);
        for(i=0;i<=count;i++){
        lqa[i]=0.0;
        fscanf(fp,"%lf",&TZ[i]);
        fscanf(fp,"%lf",&TZag2[i]);
        fscanf(fp,"%lf",&TC[i]);
        fscanf(fp,"%lf",&U[i]);
        fscanf(fp,"%lf",&Uag2[i]);
        fscanf(fp,"%lf",&T[i]);
        }
        fclose(fp);
        return;
}
```

```
/************************************************************************************
*********
*
*       File Name:      dinpl2.c
*       Purpose:        This program reads temperature and control values for two
microzone,
*                       temperature for storage, and time form input1 file. This
module
*                       is called by main function to disaggregate the second
microzone.
*
*************************************************************************************
*******/

#include "dagg8.h"
/***** get data from data-file *************/

 void gdata()
{
        int i;
        char putfile[] = "input1";
        FILE *fp;
        fp=fopen(putfile,"r");
        fscanf(fp,"%d",&count);
        for(i=0;i<=count;i++){
        lqa[i]=0.0;
        fscanf(fp,"%lf",&TZag1[i]);
        fscanf(fp,"%lf",&TZ[i]);
        fscanf(fp,"%lf",&TC[i]);
        fscanf(fp,"%lf",&Uag1[i]);
        fscanf(fp,"%lf",&U[i]);
        fscanf(fp,"%lf",&T[i]);
        }
        fclose(fp);
        return;
}
```

```
/************************************************************************************
********
*
```

```
*       File Name:      dagg21.h
*       Purpose:        This is header file for eight-zone disaggregation. It
declares
*                       variables used to disaggregate zone1 and zone2.
*
***************************************************************************************
******/

#include <stdio.h>
#include <math.h>

#defineCZ1     (double)(374.0)                            /*zone heat capacity, kw
hours/deg C*/
#defineCZ2     (double)(370.0)
#defineAZ1     (double)(410.0)
#defineAZ2     (double)(400.0)
double qz1tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
               7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
                14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;
/*lists of hourly zone cooling loads*/
double qz2tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
               7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
                14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
               35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
               35.0,35.0};
/*lists of hourly envioment temperture*/
char  outfile[] = "input312";
double pp=0.2;




/**************************************************************************************
********
*
*       File Name:      dagg22.h
*       Purpose:        This is header file for eight-zone disaggregation. It
declares
*                       variables used to disaggregate the zone3 and zone4.
*
***************************************************************************************
******/

#include <stdio.h>
#include <math.h>

#defineCZ1     (double)(300.0)
#defineCZ2     (double)(187.0)
#defineAZ1     (double)(330.0)
#defineAZ2     (double)(205.0)

double qz1tab[25] = {2.987,2.275,1.572,0.517,0.517,0.517,2.275,3.330,4.034,
               5.792,6.847,7.550,8.254,9.309,11.067,10.364,9.661,
                8.957,8.254,7.550,5.792,5.088,4.385,3.682,2.987} ;
double  qz2tab[25] = {1.841,1.489,1.127,0.758,0.208,0.208,1.237,2.017,2.846,
               3.770,4.604,05.533,06.413,07.294,08.171,09.050,08.171,
               07.292,05.533,3.770,2.891,2.544,2.192,1.841,1.841};
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
               35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
               35.0,35.0};
/*lists of hourly envioment temperture*/
char outfile[] = "input334";
double pp=0.8;
```

```
/*************************************************************************************************
********
*
*        File Name:        dagg23.h
*        Purpose:          This is header file for eight-zone disaggregation. It
declares
*                          variables used to disaggregate the zone5 and  zone6.
*
**************************************************************************************************
******/

#include <stdio.h>
#include <math.h>

#defineCZ1       (double)(374.0)
#defineCZ2       (double)(300.0)
#defineAZ1       (double)(410.0)
#defineAZ2       (double)(330.0)

double qz1tab[25] = {8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                  8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                    8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101} ;
double qz2tab[25] = {8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                  8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                    8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101} ;
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                  35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                  35.0,35.0};
/*lists of hourly envioment temperture*/
char outfile[] = "input356";
double pp=0.0;
```

```
/*************************************************************************************************
********
*
*        File Name:        dagg24.h
*        Purpose:          This is header file for eight-zone disaggregation. It
declares
*                          variables used to disaggregate the zone7 and zone8.
*
**************************************************************************************************
******/

#include <stdio.h>
#include <math.h>

#defineCZ1       (double)(187.0)
#defineCZ2       (double)(100.0)
#defineAZ1       (double)(205.0)
#defineAZ2       (double)(110.0)

double  qz1tab[25] = {5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
                  5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
                  5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223};
double  qz2tab[25] = {3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,
                  3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,
                  3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223};
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                  35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                  35.0,35.0};
/*lists of hourly envioment temperture*/
char outfile[] = "input378";
double pp=1.0;
```

```
/***********************************************************************************
*********
*
*       File Name:      dagg2.c
*                    This program computes the orginal zones optimal
*                    trajectores.
*                       The main function calls module:
*                               fix1n()
*                               fix2n()
*                               test()
*                               test1(int)
*                               plot()
*                           icd()
*                           gdata()
*
************************************************************************************
*******/

#include <math.h>
#include "dagg.h"
#include "dagg8.h"

void main()
{
        double  t1max=0.0;                              /*start at midnight, go backwards
(neg time)*/
        double  tmn;
        int     tnhour1;
        int     l, n, first1, first2, last1, last2 ;


        /**** initialize the arrays *****/

        RG1 = sqrt(G1);
        RG2 = sqrt(G2);
        RCZ1 = sqrt(CZ1);
        RCZ2 = sqrt(CZ2);
        RCC = sqrt(CC);
        for(ix1=0; ix1<=IX1MAX; ix1++)
        {       for(ix2=0; ix2<=IX2MAX; ix2++)
                {
                     cotab[ix1][ix2]=0.0;
                        ectab[ix1][ix2][0]=0.0;
                        ectab[ix1][ix2][1]=0.0;
                        for(m=0; m<LENGTH; m++)
                        {
                             t1tab[ix1][ix2][m] = 0.0+(rand()/32768.0)/1000000;
                                lqatab[ix1][ix2][m]=0.1;
                                TCtab[ix1][ix2][m] = 0.0;
                                u1tab[ix1][ix2][m] = 0.0;
                                u2tab[ix1][ix2][m] = 0.0;
                                labtab[ix1][ix2][m][0] = ix1;      /*give each node
its own lable*/
                                labtab[ix1][ix2][m][1] = ix2;
                        }
               }
       }

       /**** read data for data file ****/
        gdata();
        while( t1max > -24.0 )
       {

       /**** find the next node to update ****/

        t1max = -24.0;
        for(ix1=0; ix1<=IX1MAX; ix1++)
        {
                for(ix2=0; ix2<=IX2MAX; ix2++)
                {
                      if(t1tab[ix1][ix2][0]>t1max)
                        {       ix1u=ix1;           /*node to be updated has latest
last update*/
                                ix2u=ix2;
                        tlmax=tltab[ix1][ix2][0];
                        tlu = tlmax;
                }
        }
 }

/**** find the earliest of the admissible update times*****/

 tm = 0.0;
 for(dix2=-1; dix2<=1; dix2++)
{
         for(dix1=-1; dix1<=1; dix1++)
       {
              tn=tltab[ix1n][ix2n][0];
                 tmn = tn-DTMAX1;
              if(tmn<=tm) tm=tmn;
         }
}

/*** find the least-cost transition ***/

         tl = tltab[ix1u][ix2u][0];
         ctcmin = LARGE;
         if((tl<-23.0)||((tl<=-15.0)&&(tl>-16.0)) ||((tl<=-5.8)&&(tl>-6.0)))
         {
                tn=tl;
                dix1=0;
                dix2=0;
              price=LP;
                tnhour = tn+24;
                tme = (tn+24.0);
                tz1s=TZ1SHI;
                tz2s=TZ2SHI;
                qz1 =2.* qz1tab[tnhour]*3600;
                qz2 =2.* qz2tab[tnhour]*3600;
                TE = tetab[tnhour];
                DTMAX=24.0+tn;
                if((tl<=-15.0)&&(tl>-16.0))
                {
                tz1s=TZ1SLO;
                tz2s=TZ2SLO;
              price=HP;
                DTMAX=16.0+tl;
                }
                if((tl<=-5.8)&&(tl>-6.0))
                {
              price=HP;
                DTMAX=6.0+tl;
                }
                 dt=DTMAX;
                 ice();
                 test1(1);

         }
         else
         {
                 for(dix2=-1; dix2<=1; dix2++)
                 {
                         for(dix1=-1; dix1<=1; dix1++)
                         {
                                 tn = tltab[ix1n][ix2n][0];
                                 tnhour = tn+24;
                                 tme =tn+24.0;
                                 qz1 = qz1tab[tnhour]*3600;
                                 qz2 = qz2tab[tnhour]*3600;
                                 TE = tetab[tnhour];
                                 tz1s=TZ1SHI;
                                 tz2s=TZ2SHI;
                                 if((tme>8.0) && (tme<=18.0))
                                 {
                                         tz1s=TZ1SLO;
                                         tz2s=TZ2SLO;
                                 }
                             if(tme<=8.0)
                             price=LP;
```

```
                                     else
                                     price=HP;
                                     if((t1<=-5.0) &&(t1>-6.0))
                                     {
                                     DTMAX=0.2;
                                          for(l=1; l>=1; l--)
                                          {
                                                  dt=DTMAX/l;
                                                  if((tme>=tnhour) && (tnhour>(tme-
dt)))

                                                  {
                                                        tnhour1=24+tn-dt;
                                                        qz1=(qz1tab[tnhour1]*(tnhour-
(tme-dt))/dt+qz1tab[tnhour]

                                                        *(tme-tnhour)/dt)*3600;
                                                        qz2=(qz2tab[tnhour1]*(tnhour-
(tme-dt))/dt+qz2tab[tnhour]

                                                        *(tme-tnhour)/dt)*3600;
                                                        TE=(tetab[tnhour1]*(tnhour-
(tme-dt))/dt+tetab[tnhour]*(tme-tnhour)/dt);

                                                  }
                                                  ice();
                                             if((dix1==0) && (dix2==0) &&
(dt==DTMAX/1))

                                             {
                                             sqa=qa;
                                             sqz1=qz1;
                                             sqz2=qz2;
                                             stz1s=tz1s;
                                             stz2s=tz2s;
                                             sla=la;
                                             sprice=price;
                                             }
                                                  test1(l);
                                             }
                                          }
                                     else
                                     {
                                          DTMAX=DTMAX1;

                                          for(l=LMAX; l>=1; l--)
                                          {
                                                  dt=DTMAX/l;
                                                  if((tme>=tnhour) && (tnhour>(tme-
dt)))

                                                  {
                                                        tnhour1=24+tn-dt;
                                                        qz1=(qz1tab[tnhour1]*(tnhour-
(tme-dt))/dt+qz1tab[tnhour]

                                                        *(tme-tnhour)/dt)*3600;
                                                        qz2=(qz2tab[tnhour1]*(tnhour-
(tme-dt))/dt+qz2tab[tnhour]

                                                        *(tme-tnhour)/dt)*3600;
                                                        TE=(tetab[tnhour1]*(tnhour-
(tme-dt))/dt+tetab[tnhour]*(tme-tnhour)/dt);

                                                  }
                                                  ice();
                                             if((dix1==0) && (dix2==0)
&&(dt==DTMAX/5)){

                                             sqa=qa;
                                             sqz1=qz1;
                                             sqz2=qz2;
                                             stz1s=tz1s;
                                             stz2s=tz2s;
                                             sla=la;
                                             sprice=price;
                                             }
                                                  test1(l);
                                      }/* end of for */
                                          }/* end of else */
                                  }
                          }
                 }
```

```
                  if(tlu >= tlmax)
                  {
                          printf("\failed update");
                          break;
                  }
                  for(n=LENGTH-2; n>=0; n--)
                  {
                          TCtab[ix1u][ix2u][n+1] =
                                  TCtab[ix1next][ix2next][n];
                          u1tab[ix1u][ix2u][n+1] =
                                  u1tab[ix1next][ix2next][n];
                          u2tab[ix1u][ix2u][n+1] =
                                  u2tab[ix1next][ix2next][n];
                          tltab[ix1u][ix2u][n+1] =
                                  tltab[ix1next][ix2next][n];
                          lqatab[ix1u][ix2u][n+1] =
                                  lqatab[ix1next][ix2next][n];
                          labtab[ix1u][ix2u][n+1][0] =
                            labtab[ix1next][ix2next][n][0];
                          labtab[ix1u][ix2u][n+1][1] =
                            labtab[ix1next][ix2next][n][1];
                  }
                  cotab[ix1u][ix2u] = nco;
                if (tlu<-16.0)
                {
                ectab[ix1u][ix2u][0]=nec+ectab[ix1next][ix2next][0];
                ectab[ix1u][ix2u][1]=ectab[ix1next][ix2next][1];
                }
                else
                {
                ectab[ix1u][ix2u][1]=nec+ectab[ix1next][ix2next][1];
                ectab[ix1u][ix2u][0]=ectab[ix1next][ix2next][0];
                }
                  lqatab[ix1u][ix2u][0] = nlq;
                  if(tlu==-6.0||tlu==-16.0)
                  {
                  tltab[ix1u][ix2u][0] = tlu-(rand()/32768.0)/1000000;
                  }
                  else
                  {
                  tltab[ix1u][ix2u][0] = tlu;
                  }
                  u1tab[ix1u][ix2u][0] = uu1;
                  u2tab[ix1u][ix2u][0] = uu2;
                  TCtab[ix1u][ix2u][0] = CCT;
                  labtab[ix1u][ix2u][0][0] = ix1u;
                  labtab[ix1u][ix2u][0][1] = ix2u;
      } /*** closes the "while" loop in main() ***/
         printf("\nlabel trace");

                  for(ix2=IX2MAX; ix2>=0; ix2--)
                  {
                          for(ix1=0; ix1<=IX1MAX; ix1++)
                          {
                                  for(n=0; n<LENGTH; n++)
                                  {
                                          if(tltab[ix1][ix2][n] <= -24.0)
                                          {first1=labtab[ix1][ix2][n][0];
                                           last1=labtab[ix1][ix2][LM1][0];
                                           first2=labtab[ix1][ix2][n][1];
                                           last2=labtab[ix1][ix2][LM1][1];
                                           if(first1==last1 &&first2==last2 )
                                           {
                                           printf("\n");
                                                  for(m=0; m<LENGTH; m++)
                                                  {
                                                          printf("\nlabeltrace[%2d] =
%2d,%2d", m,

                                                          labtab[ix1][ix2][m][0],
                                                          labtab[ix1][ix2][m][1]);
                                                          printf("     tltrace[%2d] =
%6f", m,

                                                          tltab[ix1][ix2][m]+24.0);
```

```
                                                        printf("\nultrace[%2d] =
%6f", m,
                                                        ultab[ix1][ix2][m]);
                                                        printf("     u2trace[%2d] =
%6f", m,
                                                        u2tab[ix1][ix2][m]);

printf("\nlqatab=%12f",lqatab[ix1][ix2][m]);
                                                  if(tltab[ix1][ix2][m]>=0.0) break;
                                                }
                                           numb=m;

       printf("\npeak=%12f",ectab[first1][first2][1]);
                                           printf("\noff
peak=%12f",ectab[first1][first2][0]);
                                           plot();
                                        }/* end of if */
                                       }/* end of if */
                             }/* end of for */
                      }/* end of for */
               }/* end of for*/
}
/****** end of main() *****/


/*********functions to compute new-node index*************/

 int fix1n()
 {
        extern int ix1u, dix1;
       int i;
        if((ix1u<=1 && dix1==-2)||(ix1u==0 && dix1==-1)||(ix1u==IX1MAX &&
dix1==1)||(ix1u>=(IX1MAX-1) && dix1==2)) i=0;
        else i=dix1;
        return(ix1u+i);
 }

int fix2n()
{
        extern int ix2u, dix2;
        int j;
        if((ix2u==0 && dix2==-1)||(ix2u<=1 && dix2==-2)||(ix2u==IX2MAX && dix2==1)
||(ix2u>=(IX2MAX-1) && dix2==2)) j=0;
        else j=dix2;
        return(ix2u+j);
}
/************** procedure to call trial cost computation *****/

void test1(int l)
{
        v1 = (ix1n-ix1u)*DX1/dt - d1 + x1u*AZ1/CZ1;
        v2 = (ix2n-ix2u)*DX2/dt - d2 + x2u*AZ2/CZ2;
        u1 = v1/(B1*g1);
        u2 = v2/(B2*g2);
        q = (u1*g1/RCZ1 + u2*g2/RCZ2)*ACZ*dt;
        tqa = qa+lqatab[ix1n][ix2n][0]+la*pp;
        if((u1>=0.0) && (u2>=0.0) &&(q<=tqa) ){

                test();
        }
        if (((tl<-23.0)&&(tl==tlu)) ||((tl<=-15.)&&(tl>-16)&&(tl==tlu))||((tl<=-
5.8)&&(tl>-6.0)&&(tl==tlu))){
                test();
                return;
        }
        if ((((DTMAX==0.2)&&(dt==0.2))||((DTMAX==1.0)&&(dt==1.0))) && (tlu==tl) &&
(dix1==1) && (dix2==1)) {
        dix1=0;
       dix2=0;
       dt=0.2;
       tn=tltab[ix1n][ix2n][0];
       qz1=sqz1;
       qz2=sqz2;
       tz1s=stz1s;
       tz2s=stz2s;
       price=sprice;
       la=sla;
       qa=sqa;
       v1 = (ix1n-ix1u)*DX1/dt - d1 + x1u*AZ1/CZ1;
        v2 = (ix2n-ix2u)*DX2/dt - d2 + x2u*AZ2/CZ2;
        u1 = v1/(B1*g1);
        u2 = v2/(B2*g2);
        q = (u1*g1/RCZ1 + u2*g2/RCZ2)*ACZ*dt;
        tqa=qa+lqatab[ix1n][ix2n][0]+la*pp;
           test();
           dix1=1;
           dix2=1;
           return;
       }
        return;
}
/*****  procedure to compute trial cost *****/

void test()
{
        double tc, ct, ctc,tltry;
        tltry=tn-dt;
        if((tltry<-6.0)&&(tl>-5.8))return;
        if((tltry<-16.0)&&(tl>-15.0))return;
        if((tltry<-24.0)&&(tl>-23.0))return;
       if((tn>-6.0)&&(tn<-1.0)&&(((ix1u-ix1n)>0)||((ix2u-ix2n)>0)))return;
        tc = ((R1*u1*u1 + R2*u2*u2 )*price+ A1*pow(x1u-X1S,2.0)/CZ1 +
                   A2*pow(x2u-X2S,2.0)/CZ2)*dt
               + cotab[ix1n][ix2n];
        ct = (tc - cotab[ix1u][ix2u]) * (tn-dt-tm)/(tl-(tn-dt));
        ctc = tc + ct;

        if(ctc<=ctcmin)
        {
                ctcmin=ctc;
                nco=tc;
                nlq=tqa-q;
                tlu=tn-dt;
                uu1=u1;
                uu2=u2;
              CCT=CT;
                if(dt==1.0/1)nec=5*(R1*u1*u1 + R2*u2*u2)*.2;
                else if(dt==1.0/2)nec=2.5*(R1*u1*u1 + R2*u2*u2)*.2;
                else if(dt==1.0/3)nec=1.0/3/.2*(R1*u1*u1 + R2*u2*u2)*.2;
                else if(dt==1.0/4)nec=1.0/4/.2*(R1*u1*u1 + R2*u2*u2)*.2;
                else nec=(R1*u1*u1 + R2*u2*u2)*dt;
                ix1next = ix1n;
                ix2next = ix2n;
        }
        return;
}

/***** compute possible qa and CT at the time period of dt ********/

void ice()
{
       int i;
             for(i=0; i<count; i++)
             {
                   T[0]=1.0+T[0];
                 if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+1])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*dt;
                         CT=TC[i+1];
                      la=(lqa[i+1]-lqa[i])*dt/(T[i]-T[i+1]);
                         break;
                   }
                 if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+2])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-tme+dt)/dt;
                         la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                          (lqa[i+2]-lqa[i+1])*(T[i+1]-tme+dt)/(T[i+1]-T[i+2]);
                      break;
```

```
                  }
                  if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+3])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-T[i+2])
                           +U[i+3]*ACZ*(TZ[i+3]-TC[i+3])*(T[i+2]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-
T[i+2])/dt+TC[i+3]*(T[i+2]-tme+dt)/dt;
                         la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                          (lqa[i+2]-lqa[i+1])+
                          (lqa[i+3]-lqa[i+2])*(T[i+2]-tme+dt)/(T[i+2]-T[i+3]);
                      break;
                  }
                   if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+4])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-T[i+2])
                           +U[i+3]*ACZ*(TZ[i+3]-TC[i+3])*(T[i+2]-T[i+3])
                           +U[i+4]*ACZ*(TZ[i+4]-TC[i+4])*(T[i+3]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-T[i+2])/dt
                           +TC[i+3]*(T[i+2]-T[i+3])/dt
                           +TC[i+4]*(T[i+3]-tme+dt)/dt;
                         la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                          (lqa[i+2]-lqa[i+1])+
                          (lqa[i+3]-lqa[i+2])+
                          (lqa[i+4]-lqa[i+3])*(T[i+3]-tme+dt)/(T[i+3]-T[i+4]);
                      break;
                  }
                   if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+5])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-T[i+2])
                           +U[i+3]*ACZ*(TZ[i+3]-TC[i+3])*(T[i+2]-T[i+3])
                           +U[i+4]*ACZ*(TZ[i+4]-TC[i+4])*(T[i+3]-T[i+4])
                           +U[i+5]*ACZ*(TZ[i+5]-TC[i+5])*(T[i+4]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-T[i+2])/dt
                           +TC[i+3]*(T[i+2]-T[i+3])/dt
                           +TC[i+4]*(T[i+3]-T[i+4])/dt
                           +TC[i+5]*(T[i+4]-tme+dt)/dt;
                         la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                          (lqa[i+2]-lqa[i+1])+
                          (lqa[i+3]-lqa[i+2])+
                          (lqa[i+4]-lqa[i+3])+
                          (lqa[i+5]-lqa[i+4])*(T[i+4]-tme+dt)/(T[i+4]-T[i+5]);
                         break;
                  }
                   if(T[i]>=tme && tme>T[i+1] && (tme-dt)>=T[i+6])
                   {
                         qa=U[i+1]*ACZ*(TZ[i+1]-TC[i+1])*(tme-T[i+1])
                           +U[i+2]*ACZ*(TZ[i+2]-TC[i+2])*(T[i+1]-T[i+2])
                           +U[i+3]*ACZ*(TZ[i+3]-TC[i+3])*(T[i+2]-T[i+3])
                           +U[i+4]*ACZ*(TZ[i+4]-TC[i+4])*(T[i+3]-T[i+4])
                           +U[i+5]*ACZ*(TZ[i+5]-TC[i+5])*(T[i+4]-T[i+5])
                           +U[i+6]*ACZ*(TZ[i+6]-TC[i+6])*(T[i+5]-tme+dt);
                         CT=TC[i+1]*(tme-T[i+1])/dt+TC[i+2]*(T[i+1]-T[i+2])/dt
                           +TC[i+3]*(T[i+2]-T[i+3])/dt
                           +TC[i+4]*(T[i+3]-T[i+4])/dt
                           +TC[i+5]*(T[i+4]-T[i+5])/dt
                           +TC[i+6]*(T[i+5]-tme+dt)/dt;
                         la=(lqa[i+1]-lqa[i])*(tme-T[i+1])/(T[i]-T[i+1])+
                          (lqa[i+2]-lqa[i+1])+
                          (lqa[i+3]-lqa[i+2])+
                          (lqa[i+4]-lqa[i+3])+
                          (lqa[i+5]-lqa[i+4])+
                          (lqa[i+6]-lqa[i+5])*(T[i+5]-tme+dt)/(T[i+5]-T[i+6]);
                         break;
                  }
          }
}

void plot()
{
        int n;
        FILE *fp, *fopen();
        fp=fopen(outfile,"w");
        fprintf(fp,"%2d",numb);
        for(n=0; n<=numb; n++)
        {
            m=numb-n;
            fprintf(fp,"\n%6f",(X1MIN+labtab[ix1][ix2][m][0]*DX1)/RCZ1);
            fprintf(fp,"    %6f",(X2MIN+labtab[ix1][ix2][m][1]*DX2)/RCZ2);
            fprintf(fp,"    %6f",TCtab[ix1][ix2][m]);
            fprintf(fp,"\n%6f",u1tab[ix1][ix2][m]);
            fprintf(fp,"    %6f",u2tab[ix1][ix2][m]);
            fprintf(fp,"    %6f",(t1tab[ix1][ix2][m]+24.0));
            fprintf(fp,"\n%6f",lqatab[ix1][ix2][m]);
         }
        fclose(fp);
        return;

}




/**********************************************************************************
*********
*
*       File Name:      dinp21.c
*       Purpose:        This program reads temperature and control values for two
microzone,
*                       temperature for storage, and time form input21 file. This
module
*                       is called by main function to disaggregate the microzone to
zone1
*                   and zone2.
*
**********************************************************************************
*******/

#include "dagg8.h"
/***** get data from data-file *************/

 void gdata()
{
        int i;
        char putfile[] = "input21";
        FILE *fp;
        fp=fopen(putfile,"r");
        fscanf(fp,"%d",&count);
        for(i=0;i<=count;i++){
        fscanf(fp,"%lf",&TZ[i]);
        fscanf(fp,"%lf",&TZag2[i]);
        fscanf(fp,"%lf",&TC[i]);
        fscanf(fp,"%lf",&U[i]);
        fscanf(fp,"%lf",&Uag2[i]);
        fscanf(fp,"%lf",&T[i]);
        fscanf(fp,"%lf",&lqa[i]);
        }
        fclose(fp);
        return;
}




/**********************************************************************************
*********
*
*       File Name:      dinp22.c
*       Purpose:        This program reads temperature and control values for two
microzone,
*                       temperature for storage, and time form input21 file. This
module
*                       is called by main function to disaggregate the microzone to
zone3
```

```
*                         and zone4.
*
******************************************************************************************
*******/

#include "dagg8.h"
/***** get data from data-file *************/

 void gdata()
{
        int i;
        char putfile[] = "input21";
        FILE *fp;
        fp=fopen(putfile,"r");
        fscanf(fp,"%d",&count);
        for(i=0;i<=count;i++){
        fscanf(fp,"%lf",&TZag1[i]);
        fscanf(fp,"%lf",&TZ[i]);
        fscanf(fp,"%lf",&TC[i]);
        fscanf(fp,"%lf",&Uag1[i]);
        fscanf(fp,"%lf",&U[i]);
        fscanf(fp,"%lf",&T[i]);
        fscanf(fp,"%lf",&lqa[i]);
        }
        fclose(fp);
        return;
}
```

```
/*****************************************************************************************
*********
*
*       File Name:        dinp23.c
*       Purpose:          This program reads temperature and control values for two
microzone,
*                         temperature for storage, and time form input22 file. This
module
*                         is called by main function to disaggregate the microzone to
zone5
*                         and zone6.
*
******************************************************************************************
*******/

#include "dagg8.h"
/***** get data from data-file *************/

 void gdata()
{
        int i;
        char putfile[] = "input22";
        FILE *fp;
        fp=fopen(putfile,"r");
        fscanf(fp,"%d",&count);
        for(i=0;i<=count;i++){
        fscanf(fp,"%lf",&TZ[i]);
        fscanf(fp,"%lf",&TZag2[i]);
        fscanf(fp,"%lf",&TC[i]);
        fscanf(fp,"%lf",&U[i]);
        fscanf(fp,"%lf",&Uag2[i]);
        fscanf(fp,"%lf",&T[i]);
        fscanf(fp,"%lf",&lqa[i]);
        }
        fclose(fp);
        return;
}
```

```
/************************************************************************************
********
*
*       File Name:      dagg24.h
*       Purpose:        This is header file for eight-zone disaggregation. It
declares
*                       variables used to disaggregate the zone7 and zone8.
*
*************************************************************************************
******/

#include <stdio.h>
#include <math.h>

#defineCZ1    (double)(187.0)
#defineCZ2    (double)(100.0)
#defineAZ1    (double)(205.0)
#defineAZ2    (double)(110.0)

double  qz1tab[25] = {5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
                5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
                5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223};
double  qz2tab[25] = {3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,
                3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223,
                3.223,3.223,3.223,3.223,3.223,3.223,3.223,3.223};
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                35.0,35.0};
/*lists of hourly enviioment temperture*/
char outfile[] = "input378";
double pp=1.0;
```

A.3 Source Code for Four-Zone Cooling System

```
CC = gcc
FLAGS = -ansi -O
LIBFLAGS = -lm

all: agg4 block4

agg4: agg4.o
	$(CC) agg4.o -o agg4 $(LIBFLAGS) -g

block4: block4.o
	$(CC) block4.o -o block4 $(LIBFLAGS) -g

agg4.o: agg4.c
	$(CC) $(FLAGS) -g -c agg4.c

block4.o: block4.c
	$(CC) $(FLAGS) -c -g block4.c
```

```
/**********************************************************************************
*********
*
*       File Name:      agg4.c
*       Purpose:        This program computes the optimal trajectory for two
macrozones
*                       and storage. The main function calls module:
*                               fix1n()
*                               fix2n()
*                               fix3n()
*                               test()
*                               test1(int)
*                               plot()
*
***********************************************************************************
*******/


#include <stdio.h>
#include <math.h>

#define LENGTH(int)160                           /*max length of trajectories*/
#defineLM1     (int)(LENGTH-1)
#define LARGE  (double)1000000000.0
#defineLP       (double)0.5                       /*off peak-hour energy price
ratio*/
#defineHP       (double)1.0                       /* peak-hour energy price ratio*/
#defineTZ1SLO (double)24.0                        /*setpoint temp for occupied
zone*/
#defineTZ1SHI (double)30.0                        /*setpoint temp for unoccupied
zone*/
#defineTZ2SLO (double)24.0
#defineTZ2SHI (double)30.0
#defineTZ1MIN (double)(TZ1SLO-1.0)                 /*min zone temperature
constraint*/
#defineIX1SPAN(int)9                             /*number of discrete points per
dim.*/
#defineIX1MAX (int)(IX1SPAN-1)
#defineTZ1MAX (double)(TZ1SHI+1.0)                 /*max zone temperature
constraint*/
#defineTZ2MIN (double)(TZ2SLO-1.0)
#defineIX2SPAN(int)9
#defineIX2MAX (int)(IX2SPAN-1)
```

```
#defineTZ2MAX (double)(TZ2SHI+1.0)
#defineTCMIN  (double)8.0                         /*min tank temperature constraint*/
#defineIX3SPAN(int)15
#defineIX3MAX (int)(IX3SPAN-1)
#defineTCMAX  (double)15.0                         /*max tank temperature
constraint*/
#defineT0     (double)(35.0-20.0)                        /*coolant temperature for
condenser*/
#defineA1     (double)1000.0                      /*cost parameter, zone temp
deviation*/
#defineA2     (double)1000.0
#defineR1     (double)0.00003                            /*cost parameter,
circulation pump op.*/
#defineR2     (double)0.00003
#defineR3     (double)1.0
#defineCZ1    (double)(374.+187.)                 /*microzone heat capacity, kw
hours/deg C*/
#defineCZ2    (double)(300.0+187.)
#defineCC     (double)90000.                      /*tank heat capacity, kw hours/deg
C*/
#defineG2     (double)CZ2/CC
#defineG1     (double)CZ1/CC
#defineAZ1    (double)(410.0+205.0)               /*heat transfer coef for
microzone*/
#defineAZ2    (double)(330.0+205.0)
#defineAC     (double)17.6                        /*heat transfer coef, tank*/
#define ACZ   (double)0.75                        /*heat exchanger coef*/
#defineB1     (double)(-ACZ/CZ1)
#defineB2     (double)(-ACZ/CZ2)
#defineTMAX   (double)20.0                         /*max temp diff, refrig unit*/
#defineCOPMAX (double)4.0                         /*coef of performance, refrig
unit*/
#defineCOPM1  (double)(COPMAX-1.0)
#defineBETA   (double)(COPM1/(CC*TMAX))
#defineC33    (double)(-COPM1*(1.0 - T0/TMAX)/RCC)
#defineB3     (double)(-BETA)
#defineDTMAX1 (double)1.0                         /*max time-step allowed when time
is not smaller than -23.0*/
#defineLMAX   (int)5                              /*number of control amplitudes
tested*/
#defineX1MIN  (double)(RCZ1*TZ1MIN)                /*min value, normalized state
variable*/
#defineX1MAX  (double)(RCZ1*TZ1MAX)                /*max value, normalized state
variable*/
#defineDX1    (double)((X1MAX-X1MIN)/IX1MAX)            /*stepsize, normalized
state var*/
#defineX2MIN  (double)(RCZ2*TZ2MIN)
#defineX2MAX  (double)(RCZ2*TZ2MAX)
#defineDX2    (double)((X2MAX-X2MIN)/IX2MAX)
#defineX3MIN  (double)(RCC*TCMIN)
#defineX3MAX  (double)(RCC*TCMAX)
#defineDX3    (double)((X3MAX-X3MIN)/IX3MAX)
#defineX1S    (double)(RCZ1*tz1s)                 /*desired value, normalized state
var*/
#defineX2S    (double)(RCZ2*tz2s)
#definex1u    (double)(X1MIN + ix1u*DX1)           /*discrete level, node being
updated*/
#definex2u    (double)(X2MIN + ix2u*DX2)
#definex3u    (double)(X3MIN + ix3u*DX3)
#defineg1     (double)(x1u-RG1*x3u)
#defineg2     (double)(x2u-RG2*x3u)
#defineg3     (double)x3u
#defined1     (double)((qz1+AZ1*35.0)/RCZ1)             /*normalized zone cooling
load*/
#defined2     (double)((qz2+AZ2*35.0)/RCZ2)
#defined3     (double)(AC*35.0/RCC)
#define ix1n  (int)fix1n()                         /*index of next node examined*/
#defineix2n   (int)fix2n()
#defineix3n   (int)fix3n()
#definex1n    (double)(X1MIN + ix1n*DX1)           /*discrete level, next node
examined*/
#definex2n    (double)(X2MIN + ix2n*DX2)
#definex3n    (double)(X3MIN + ix3n*DX3)
```

```
#define
labtab(i,j,k,l,m)(*(LABTAB+(i)*IX2SPAN*IX3SPAN*LENGTH*3+(j)*IX3SPAN*LENGTH*3+(k)*LENG
TH*3+(l)*3+(m)))
#define cotab(i,j,k)(*(COTAB+(i)*IX2SPAN*IX3SPAN+(j)*IX3SPAN+(k)))
#define ectab(i,j,k,l)(*(ECTAB+(i)*IX2SPAN*IX3SPAN*2+(j)*IX3SPAN*2+(k)*2+(l)))
#define
tltab(i,j,k,l)(*(TLTAB+(i)*IX2SPAN*IX3SPAN*LENGTH+(j)*IX3SPAN*LENGTH+(k)*LENGTH+(l)))

/*********** function declarations*************/

void main(void);
int  fix1n(void);
int  fix2n(void);
int  fix3n(void);
void test(void);
void test1(int);
void plot(void);


/************ global variables ****************/
int     T[4][LENGTH];                                          /*table of
temperature*/
double  COP[LM1];                                              /*table of COP*/
double *COTAB, *ECTAB, *TLTAB;
unsigned char *LABTAB;
int     ix1u,ix2u,ix3u,ix1,ix2,ix3,m,count;
int     dix1,dix2,dix3,ix1next,ix2next,ix3next;
double
        pu1,pu2,pu3,pu4,u1,u2,u3,v1,v2,v3,RG1,RG2,RCZ1,RCZ2,RCC,qz1,qz2,pqz1,pqz2,pqz3
,pqz4,tz1s,tz2s,DTMAX;
double tn,tm,dt,tl,uu1,uu2,uu3,tlu,nco,ctcmin,r1,r2,r3,a1,a2,nec,price,TE;
double qz1tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
              7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
               14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;
/*lists of hourly zone cooling loads*/
double  qz2tab[25] = {8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
              8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
              8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101};
double  qz3tab[25] = {1.841,1.489,1.127,0.758,0.208,0.208,1.237,2.017,2.846,
              3.770,4.604,05.533,06.413,07.294,08.171,09.050,08.171,
              07.292,05.533,3.770,2.891,2.544,2.192,1.841,1.841};
double  qz4tab[25] = {5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
              5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
              5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223};
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
              35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
              35.0,35.0};
/*lists of hourly envioment temperture*/

/*********** main program *****************/

void main()
{
        double tlmax=0.0;                       /*start at midnight, go backwards (neg
time)*/
        double  tmn,time;
        int     tnhour,tnhour1;
        int     l, n, first1, first2, first3, last1, last2, last3;
        if((LABTAB=(unsigned char
*)malloc(IX1SPAN*IX2SPAN*IX3SPAN*LENGTH*3*sizeof(unsigned char)))==NULL){
        printf(stderr,"error allocating sufficient memory...aborting\n");
        exit(-1);
        }
        if((COTAB=(double *)malloc(IX1SPAN*IX2SPAN*IX3SPAN*sizeof(double)))==NULL){
        printf(stderr,"error allocating sufficient memory...aborting\n");
        exit(-1);
        }
        if((ECTAB=(double *)malloc(IX1SPAN*IX2SPAN*IX3SPAN*2*sizeof(double)))==NULL){
        printf(stderr,"error allocating sufficient memory...aborting\n");
        exit(-1);
        }
        if((TLTAB=(double
*)malloc(IX1SPAN*IX2SPAN*IX3SPAN*LENGTH*sizeof(double)))==NULL){
        printf(stderr,"error allocating sufficient memory...aborting\n");
        exit(-1);
        }


        /**** initialize the arrays *****/

        RG1 = sqrt(G1);
        RG2 = sqrt(G2);
        RCZ1 = sqrt(CZ1);
        RCZ2 = sqrt(CZ2);
        RCC = sqrt(CC);
        for(n=0; n<4; n++)
         {        for(m=0; m<LENGTH; m++)
                {
                 T[n][m]=0.0;
                 COP[m]= 0.0;
                 }
         }
        for(ix1=0; ix1<=IX1MAX; ix1++)
        {       for(ix2=0; ix2<=IX2MAX; ix2++)
                {
                        for(ix3=0; ix3<=IX3MAX; ix3++)
                        {
                                cotab(ix1,ix2,ix3) = 0.0;
                                ectab(ix1,ix2,ix3,0) = 0.0;
                                ectab(ix1,ix2,ix3,1) = 0.0;
                                for(m=0; m<LENGTH; m++)
                                {
                                        tltab(ix1,ix2,ix3,m) =
0.0+(rand()/32768.0)/1000000;
                                        labtab(ix1,ix2,ix3,m,0) = ix1;      /*give each
node its own lable*/
                                        labtab(ix1,ix2,ix3,m,1) = ix2;
                                        labtab(ix1,ix2,ix3,m,2) = ix3;
                                }
                        }
                }
        }
        while( tlmax > -24.0 )
{

/**** find the next node to update ****/

        tlmax = -24.0;
        for(ix1=0; ix1<=IX1MAX; ix1++)
        {
                for(ix2=0; ix2<=IX2MAX; ix2++)
                {
                        for(ix3=0; ix3<=IX3MAX; ix3++)
                        {
                                if(tltab(ix1,ix2,ix3,0)>tlmax)
                                {       ix1u=ix1;           /*node to be updated has
latest last update*/
                                        ix2u=ix2;
                                        ix3u=ix3;
                                        tlmax=tltab(ix1,ix2,ix3,0);
                                        tlu = tlmax;
                                }
                        }
                }
        }

/**** find the earliest of the admissible update times*****/

        tm = 0.0;
                for(dix3=-1; dix3<=1; dix3++)
                {
                        for(dix2=-1; dix2<=1; dix2++)
                        {
                                for(dix1=-1; dix1<=1; dix1++)
                                {
                                        tn=tltab(ix1n,ix2n,ix3n,0);
                                          tmn = tn-DTMAX1;
                                        if(tmn<=tm) tm=tmn;
                                }
                }
        }

/*** find the least-cost transition ***/

        tl = tltab(ix1u,ix2u,ix3u,0);
        ctcmin = LARGE;
        if((tl<-23.0) || ((tl<=-15.0)&&(tl>-16.0)) ||((tl<=-5.0) && (tl>-6.0)))
        {
                 tn=tl;
                 dix1=0;
                 dix2=0;
                 dix3=0;
                 price=LP;
                tnhour = tn+24;
                time = tn+24.0;
                tz1s=TZ1SHI;
                tz2s=TZ2SHI;
                r1=R1;
                r2=R2;
                r3=R3;
                a1=A1;
                a2=A2;
                qz1 =(qz1tab[tnhour]+qz3tab[tnhour])*3600;
                qz2 =(qz2tab[tnhour]+qz4tab[tnhour])*3600;
                pqz1=qz1tab[tnhour]*3600;
                pqz2=qz2tab[tnhour]*3600;
                pqz3=qz3tab[tnhour]*3600;
                pqz4=qz4tab[tnhour]*3600;
                TE = tetab[tnhour];
                DTMAX=24.0+tn;
                if((tl<=-15.0)&&(tl>-16.0))
                {
                tz1s=TZ1SLO;
                tz2s=TZ2SLO;
                price=HP;
                DTMAX=16.0+tl;
                }
                if((tl<=-5.0)&&(tl>-6.0))
                {
                price=HP;
                DTMAX=6.0+tl;
                }
                dt=DTMAX;
                test1(1);

          }
        else
        {
        for(dix3=-1; dix3<=1; dix3++)
        {
                for(dix2=-1; dix2<=1; dix2++)
                {
                        for(dix1=-1; dix1<=1; dix1++)
                        {
                                tn = tltab(ix1n,ix2n,ix3n,0);
                                tnhour = tn+24;
                                time = tn+24.0;
                                  qz1 =(qz1tab[tnhour]+qz3tab[tnhour])*3600;
                                  qz2 =(qz2tab[tnhour]+qz4tab[tnhour])*3600;
                                  pqz1=qz1tab[tnhour]*3600;
                                  pqz2=qz2tab[tnhour]*3600;
                                  pqz3=qz3tab[tnhour]*3600;
                                  pqz4=qz4tab[tnhour]*3600;
                                TE = tetab[tnhour];
                                tz1s=TZ1SHI;
                                tz2s=TZ2SHI;
                                r1=R1;
                                r2=R2;
                                r3=R3;
                                a1=A1;
                                a2=A2;
                                if((time>8.0) && (time<=18.0))
                                {
                                        tz1s=TZ1SLO;
```

```
                                           tz2s=TZ2SLO;
                                    }
                                    if(time<=8.0)
                                    price=LP;
                                    else
                                    price=HP;
                                    DTMAX=DTMAX1;

                                    for(l=LMAX; l>=1; l--)
                                    {
                                           dt=DTMAX/l;
                                           if((time>=tnhour) && (tnhour>(time-dt)))
                                           {
                                                 tnhour1=24+tn-dt;

qz1=((qz1tab[tnhour1]+qz3tab[tnhour1])*(tnhour-(time-
dt))/dt+(qz1tab[tnhour]+qz3tab[tnhour])
                                                 *(time-tnhour)/dt)*3600;


qz2=((qz2tab[tnhour1]+qz4tab[tnhour1])*(tnhour-(time-
dt))/dt+(qz2tab[tnhour]+qz4tab[tnhour])
                                                 *(time-tnhour)/dt)*3600;
                                                 pqz1=(qz1tab[tnhour1]*(tnhour-(time-
dt))/dt+qz1tab[tnhour]
                                                 *(time-tnhour)/dt)*3600;
                                                 pqz2=(qz2tab[tnhour1]*(tnhour-(time-
dt))/dt+qz2tab[tnhour]
                                                 *(time-tnhour)/dt)*3600;
                                                 pqz3=(qz3tab[tnhour1]*(tnhour-(time-
dt))/dt+qz3tab[tnhour]
                                                 *(time-tnhour)/dt)*3600;
                                                 pqz4=(qz4tab[tnhour1]*(tnhour-(time-
dt))/dt+qz4tab[tnhour]
                                                 *(time-tnhour)/dt)*3600;
                                                 TE=(tetab[tnhour1]*(tnhour-(time-
dt))/dt+tetab[tnhour]*(time-tnhour)/dt);

                                                }
                                           test1(l);
                                    }
                            }
                     }
              }
              }
              if(tlu >= tlmax)
              {
                     printf("\failed update");
                     break;
              }
              for(n=LENGTH-2; n>=0; n--)
              {
                     tltab(ix1u,ix2u,ix3u,n+1) =
                            tltab(ix1next,ix2next,ix3next,n);
                     labtab(ix1u,ix2u,ix3u,n+1,0) =
                       labtab(ix1next,ix2next,ix3next,n,0);
                     labtab(ix1u,ix2u,ix3u,n+1,1) =
                       labtab(ix1next,ix2next,ix3next,n,1);
                     labtab(ix1u,ix2u,ix3u,n+1,2) =
                       labtab(ix1next,ix2next,ix3next,n,2);
              }
              cotab(ix1u,ix2u,ix3u) = nco;
              if (tlu<-16.0)
              {
                      ectab(ix1u,ix2u,ix3u,0) = nec+ectab(ix1next,ix2next,ix3next,0);
                      ectab(ix1u,ix2u,ix3u,1) = ectab(ix1next,ix2next,ix3next,1);
              }
              else
              {
                      ectab(ix1u,ix2u,ix3u,1) = nec+ectab(ix1next,ix2next,ix3next,1);
                      ectab(ix1u,ix2u,ix3u,0) = ectab(ix1next,ix2next,ix3next,0);
              }
              if(tlu==-6.0||tlu==-16.0)
              {
              tltab(ix1u,ix2u,ix3u,0) = tlu-(rand()/32768.0)/1000000;
```

```
                }
                else
                {
                tltab(ix1u,ix2u,ix3u,0) = tlu;
                }
                labtab(ix1u,ix2u,ix3u,0,0) = ix1u;
                labtab(ix1u,ix2u,ix3u,0,1) = ix2u;
                labtab(ix1u,ix2u,ix3u,0,2) = ix3u;
        } /*** closes the "while" loop in main() ***/

        printf("\nLABLE TRACES");
        for(ix3=IX3MAX; ix3>=0; ix3--)
        {
                for(ix2=IX2MAX; ix2>=0; ix2--)
                {
                        for(ix1=0; ix1<=IX1MAX; ix1++)
                        {
                                for(n=0; n<LENGTH; n++)
                                {
                                        if(tltab(ix1,ix2,ix3,n) <= -24.0)
                                        {first1=labtab(ix1,ix2,ix3,n,0);
                                         last1=labtab(ix1,ix2,ix3,LM1,0);
                                         first2=labtab(ix1,ix2,ix3,n,1);
                                         last2=labtab(ix1,ix2,ix3,LM1,1);
                                         first3=labtab(ix1,ix2,ix3,n,2);
                                         last3=labtab(ix1,ix2,ix3,LM1,2);
                                         if(first1==last1 && first2==last2 &&
first3==last3)

                                        {
                                        printf("\n");
                                                for(m=0; m<LENGTH; m++)
                                                {
                                                        T[1][m]=labtab(ix1,ix2,ix3,m,0);
                                                        T[2][m]=labtab(ix1,ix2,ix3,m,1);
                                                        T[3][m]=labtab(ix1,ix2,ix3,m,2);
                                                        COP[m]=(COPMAX-1)*(1-(T0-
T[3][m])/TMAX);

                                                        printf("\nlabeltrace[%2d] =
%2d,%2d,%2d", m,

                                                        labtab(ix1,ix2,ix3,m,0),
                                                        labtab(ix1,ix2,ix3,m,1),
                                                        labtab(ix1,ix2,ix3,m,2));
                                                        printf("\ttltrace[%2d] = %6f", m,
                                                        tltab(ix1,ix2,ix3,m));
                                                              if(tltab(ix1,ix2,ix3,m)>=0.0)
                                                              break;
                                                }
                                                     count=m;
                                                printf("\npeak-hour energy cost = %12f",
                                                ectab(first1,first2,first3,1));
                                                printf("\noff peak-hour energy cost =
%12f",

                                                ectab(first1,first2,first3,0));
                                              printf("\n");
                                        plot();
                                        }
                                    }
                                }
                        }
                }
        }
}
/****** end of main() *****/


/*********functions to compute new-node index*************/

int fix1n()
{
        extern int ix1u, dix1;
        int i;
        if((ix1u==0 && dix1<0)||(ix1u==IX1MAX && dix1>0)) i=0;
        else i=dix1;
        return(ix1u+i);
}
```

```
int fix2n()
{
        extern int ix2u, dix2;
        int j;
        if((ix2u==0 5& dix2<0)||(ix2u==IX2MAX && dix2>0)) j=0;
        else j=dix2;
        return(ix2u+j);
}

int fix3n()
{
        extern int ix3u, dix3;
        int k;
        if((ix3u==0 && dix3<0)||(ix3u==IX3MAX && dix3>0)) k=0;
        else k=dix3;
        return(ix3u+k);
}


/************* procedure to call trial cost computation *****/

void test1(int l)
{
        v1 = (ix1n-ix1u)*DX1/dt - d1 + x1u*AZ1/CZ1;
        v2 = (ix2n-ix2u)*DX2/dt - d2 + x2u*AZ2/CZ2;
        v3 = (ix3n-ix3u)*DX3/dt - d3 + x3u*AC/CC;
        u1 = v1/(B1*g1);
        u2 = v2/(B2*g2);
        u3 = (RG1*v1 + RG2*v2 + v3)/(C33 + B3*g3);
        if((u1>=0.0) && (u2>=0.0) && (u3>=0.0))
        {
                test();
        }
        return;
}


/*****  procedure to compute trial cost *****/

void test()
{
        double tc, ct, ctc,t1try;
        t1try=tn-dt;
        if((t1try<-6.0)&&(t1>-5.0))return;
        if((t1try<-16.0)&&(t1>-15.0))return;
        if((t1try<-24.0)&&(t1>-23.0))return;
        pu1=u1*pqz1/qz1, pu3=u1*pqz3/qz1;
        pu2=u2*pqz2/qz2; pu4=u2*pqz4/qz2;
        tc = ((r1*pu1*pu1+r1*pu3*pu3 + r2*pu2*pu2 + r2*pu4*pu4+r3*u3 )*price+
a1*pow(x1u-X1S,2.0)/CZ1 +
                   a2*pow(x2u-X2S,2.0)/CZ2)*dt
                 + cotab(ix1n,ix2n,ix3n);
        ct = (tc - cotab(ix1u,ix2u,ix3u)) * (tn-dt-tm)/(t1-(tn-dt));
        ctc = tc + ct;
        if(ctc<=ctcmin)
        {
                ctcmin=ctc;
                nco=tc;
                t1u=tn-dt;
                uu1=u1;
                uu2=u2;
                uu3=u3;
                nec=(r1*pu1*pu1 + r1*pu3*pu3 + r2*pu2*pu2 +r2*pu4*pu4 + r3*u3)*dt;
                ix1next = ix1n;
                ix2next = ix2n;
                ix3next = ix3n;
        }
        return;
}


/***************output time and u1 in plotU1 file*****************/
```

```
void plot()
{
         int n;
         FILE *fp, *fopen();
        fp=fopen("input","w");
         fprintf(fp,"%2d",count);
        for(n=0; n<=count; n++)
        {
                m=count-n;
               fprintf(fp,"\n%2d",T[1][m]);
               fprintf(fp,"   %2d",T[2][m]);
               fprintf(fp,"   %2d",T[3][m]);
               fprintf(fp,"   %6f",(t1tab(ix1,ix2,ix3,m)));
         }
        fclose(fp);
        return;
}
```

```
/***************************************************************************************
*********
*
*
*       File Name:     block4.c
*       Purpose:       This function using block by block method to compute temperature
*                      trajectories for four-zone cooling system. It includes function:
*                              main()
*                              fix1n()
*                              fix2n()
*                              fix3n()
*                              fix4n()
*                              fix5n()
*                              calcu()
*                              test1()
*                              test(int)
*                              trajy()
*
****************************************************************************************
********/
#include <stdio.h>
#include <math.h>

#define LENGTH(int)120                                 /*max length of trajectories*/
#defineLM1     (int)(LENGTH-1)
#define LARGE  (double)1000000000.0
#defineLP       (double)0.5                             /*off peak-hour energy price
ratio*/
#defineHP       (double)1.0                             /* peak-hour energy price ratio*/
#defineTZ1SLO (double)24.0                              /*setpoint temp for occupied
zone*/
#defineTZ1SHI (double)30.0                              /*setpoint temp for unoccupied
zone*/
#defineTZ2SLO (double)24.0
#defineTZ2SHI (double)30.0
#defineTZ3SLO (double)24.0
#defineTZ3SHI (double)30.0
#defineTZ4SLO (double)24.0
#defineTZ4SHI (double)30.0
#defineTZ1MIN (double)(TZ1SLO-1.0)                       /*min zone temperature
constraint*/
#defineTZ1MAX (double)(TZ1SHI+1.0)                       /*max zone temperature
constraint*/
#define IX1SPAN (int)7
#define HIX1SPAN (int)9                                /*number of discrete points per dim
*/
#defineTZ2MIN (double)(TZ2SLO-1.0)
#defineTZ2MAX (double)(TZ2SHI+1.0)
#define IX2SPAN (int)7
#define HIX2SPAN (int)9
#defineTZ3MIN (double)(TZ2SLO-1.0)
```

```
#defineTZ3MAX (double)(TZ2SHI+1.0)
#define IX3SPAN (int)7
#define HIX3SPAN (int)9
#defineTZ4MIN (double)(TZ2SLO-1.0)
#defineTZ4MAX (double)(TZ2SHI+1.0)
#define IX4SPAN (int)7
#define HIX4SPAN (int)9
#defineTCMIN  (double)8.0                         /*min tank temperature constraint*/
#defineTCMAX  (double)15.0                         /*max tank temperature
constraint*/
#define IX5SPAN (int)7
#define HIX5SPAN (int)15
#defineT0      (double)(35.0-20.0)                         /*coolant temperature for
condenser*/
#defineA1      (double)10000.0                             /*cost parameter, zone temp
deviation*/
#defineA2      (double)10000.0
#defineA3      (double)10000.0
#defineA4      (double)10000.0
#defineR1      (double)0.00003                              /*cost parameter,
circulation pump op.*/
#defineR2      (double)0.00003
#defineR3      (double)0.00003
#defineR4      (double)0.00003
#defineCZ1     (double)(374.)                     /*zone heat capacity, kw hours/deg
C*/
#defineCZ2     (double)(187.0)
#defineCZ3     (double)(300.0)
#defineCZ4     (double)(187.0)
#defineCC      (double)85000.                     /*tank heat capacity, kw hours/deg
C*/
#defineG4      (double)CZ4/CC
#defineG3      (double)CZ3/CC
#defineG2      (double)CZ2/CC
#defineG1      (double)CZ1/CC
#defineAZ1     (double)(410.0)                             /*heat transfer coef for
zone*/
#defineAZ2     (double)(205.0)
#defineAZ3     (double)(330.0)
#defineAZ4     (double)(205.0)
#defineAC      (double)17.6                       /*heat transfer coef, tank*/
#define ACZ    (double)0.75                       /*heat exchanger coef*/
#defineB1      (double)(-ACZ/CZ1)
#defineB2      (double)(-ACZ/CZ2)
#defineB3      (double)(-ACZ/CZ3)
#defineB4      (double)(-ACZ/CZ4)
#defineTMAX    (double)20.0                        /*max temp diff, refrig unit*/
#defineCOPMAX  (double)4.0                        /*coef of performance, refrig
unit*/
#defineCOPM1   (double)(COPMAX-1.0)
#defineBETA    (double)(COPM1/(CC*TMAX))
#defineC55     (double)(-COPM1*(1.0 - T0/TMAX)/RCC)
#defineB5      (double)(-BETA)
#defineDTMAX1  (double)0.5                        /*max time-step allowed when time
is not smaller than -23.0*/
#defineLMAX    (int)5                             /*number of control amplitudes
tested*/
#defineX1MIN   (double)(RCZ1*TZ1MIN)               /*min value, normalized state
variable*/
#defineX1MAX   (double)(RCZ1*TZ1MAX)               /*max value, normalized state
variable*/
#defineDX1     (double)((X1MAX-X1MIN)/(HIX1SPAN-1))              /*stepsize,
normalized state var*/
#defineX2MIN   (double)(RCZ2*TZ2MIN)
#defineX2MAX   (double)(RCZ2*TZ2MAX)
#defineDX2     (double)((X2MAX-X2MIN)/(HIX2SPAN-1))
#defineX3MIN   (double)(RCZ3*TZ3MIN)
#defineX3MAX   (double)(RCZ3*TZ3MAX)
#defineDX3     (double)((X3MAX-X3MIN)/(HIX3SPAN-1))
#defineX4MIN   (double)(RCZ4*TZ4MIN)
#defineX4MAX   (double)(RCZ4*TZ4MAX)
#defineDX4     (double)((X4MAX-X4MIN)/(HIX4SPAN-1))
#defineX5MIN   (double)(RCC*TCMIN)
#defineX5MAX   (double)(RCC*TCMAX)
#defineDX5     (double)((X5MAX-X5MIN)/(HIX5SPAN-1))
```

```
#defineX1S     (double)(RCZ1*tz1s)                   /*desired value, normalized state
var*/
#defineX2S     (double)(RCZ2*tz2s)
#defineX3S     (double)(RCZ3*tz3s)
#defineX4S     (double)(RCZ4*tz4s)
#definex1u     (double)(X1MIN + ix1u*DX1)             /*discrete level, node being
updated*/
#definex2u     (double)(X2MIN + ix2u*DX2)
#definex3u     (double)(X3MIN + ix3u*DX3)
#definex4u     (double)(X4MIN + ix4u*DX4)
#definex5u     (double)(X5MIN + ix5u*DX5)
#defineg1      (double)(x1u-RG1*x5u)
#defineg2      (double)(x2u-RG2*x5u)
#defineg3      (double)(x3u-RG3*x5u)
#defineg4      (double)(x4u-RG4*x5u)
#defineg5      (double)x5u
#defined1      (double)((qz1+AZ1*35.0)/RCZ1)                /*normalized zone cooling
load*/
#defined2      (double)((qz2+AZ2*35.0)/RCZ2)
#defined3      (double)((qz3+AZ3*35.0)/RCZ3)
#defined4      (double)((qz4+AZ4*35.0)/RCZ4)
#defined5      (double)(AC*35.0/RCC)
#define ix1n   (int)fix1n()                           /*index of next node examined*/
#defineix2n    (int)fix2n()
#defineix3n    (int)fix3n()
#defineix4n    (int)fix4n()
#defineix5n    (int)fix5n()
#definex1n     (double)(X1MIN + ix1n*DX1)             /*discrete level, next node
examined*/
#definex2n     (double)(X2MIN + ix2n*DX2)
#definex3n     (double)(X3MIN + ix3n*DX3)
#definex4n     (double)(X4MIN + ix4n*DX4)
#definex5n     (double)(X5MIN + ix5n*DX5)
#define
tltab(i,j,k,l,m,n)(*(TLTAB+(i)*IX2SPAN*IX3SPAN*IX4SPAN*IX5SPAN*LENGTH+(j)*IX3SPAN*IX4
SPAN*IX5SPAN*LENGTH+(k)*IX4SPAN*IX5SPAN*LENGTH+(l)*IX5SPAN*LENGTH+(m)*LENGTH+(n)))
#define
labtab(i,j,k,l,m,n,o)(*(LABTAB+(i)*IX2SPAN*IX3SPAN*IX4SPAN*IX5SPAN*LENGTH*5+(j)*IX3SP
AN*IX4SPAN*IX5SPAN*LENGTH*5+(k)*IX4SPAN*IX5SPAN*LENGTH*5+(l)*IX5SPAN*LENGTH*5+(m)*LEN
GTH*5+(n)*5+(o)))
#define
cotab(i,j,k,l,m)(*(COTAB+(i)*IX2SPAN*IX3SPAN*IX4SPAN*IX5SPAN+(j)*IX3SPAN*IX4SPAN*IX5S
PAN+(k)*IX4SPAN*IX5SPAN+(l)*IX5SPAN+(m)))
#define
ectab(i,j,k,l,m,n)(*(ECTAB+(i)*IX2SPAN*IX3SPAN*IX4SPAN*IX5SPAN*2+(j)*IX3SPAN*IX4SPAN*
IX5SPAN*2+(k)*IX4SPAN*IX5SPAN*2+(l)*IX5SPAN*2+(m)*2+(n)))

/*********** function declarations*************/

void main(void);
int  fix1n(void);
int  fix2n(void);
int  fix3n(void);
int  fix4n(void);
int  fix5n(void);
void calcu(void);
void test(void);
void test1(int);
void trajy(void);


/************ global variables ****************/

double *COTAB,*ECTAB,*TLTAB;
unsigned char *LABTAB;
int ix1u,ix2u,ix3u,ix4u,ix5u,ix1,ix2,ix3,ix4,ix5,n=0,m=0,count;
int dix1,dix2,dix3,dix4,dix5,ix1next,ix2next,ix3next,ix4next,ix5next,dir1=0,
dir2=0,dir3=0;
int
IX1MIN,IX1MAX,IX2MIN,IX2MAX,IX3MIN,IX3MAX,IX1f,IX2f,IX3f,IX1l,IX2l,IX3l,Tz1[120],Tz2[
120],Tc[120];
int first1, first2, first3, first4,first5,last1, last2, last3,last4,last5;
double u1,u2,u3,u4,u5,v1,v2,v3,v4,v5,RG1,RG2,RG3,RG4,RCZ1,RCZ2,RCZ3,RCZ4,RCC;
double qz1,qz2,qz3,qz4,tz1s,tz2s,tz3s,tz4s,T[400];
double tn,tm,dt,tl,tlu,nco,ctcmin,nec,price,TE;
double qz1tab[25] = {3.682,2.978,2.275,1.517,0.517,0.517,2.275,4.034,5.792,
                  7.550,9.309,11.067,12.826,14.584,16.343,18.101,16.343,
                   14.584,11.067,7.550,5.792,5.088,4.385,3.682,3.682} ;
/*lists of hourly zone cooling loads*/
double  qz2tab[25] = {1.841,1.489,1.127,0.758,0.208,0.208,1.237,2.017,2.846,
                  3.770,4.604,05.533,06.413,07.294,08.171,09.050,08.171,
                  07.292,05.533,3.770,2.891,2.544,2.192,1.841,1.841};
double  qz3tab[25] = {8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                  8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101,
                  8.101,8.101,8.101,8.101,8.101,8.101,8.101,8.101};
double  qz4tab[25] = {5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
                  5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223,
                  5.223,5.223,5.223,5.223,5.223,5.223,5.223,5.223};
double  tetab[25] = {35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                  35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,35.0,
                  35.0,35.0};
/*lists of hourly enviomient temperture*/

/*********** main program *****************/
void main()
{
        int  i;
        if((TLTAB=(double
*)malloc(IX1SPAN*IX2SPAN*IX3SPAN*IX4SPAN*IX5SPAN*LENGTH*sizeof(double)))==NULL){
        printf(stderr,"Error allocating sufficient memory...aborting\n");
        exit(-1);
        }
        if((LABTAB=(unsigned char
*)malloc(IX1SPAN*IX2SPAN*IX3SPAN*IX4SPAN*IX5SPAN*LENGTH*5*sizeof(unsigned
char)))==NULL){
        printf(stderr,"Error allocating sufficient memory...aborting\n");
        exit(-1);
        }
        if((COTAB=(double
*)malloc(IX1SPAN*IX2SPAN*IX3SPAN*IX4SPAN*IX5SPAN*sizeof(double)))==NULL){
        printf(stderr,"Error allocating sufficient memory...aborting\n");
        exit(-1);
        }
        if((ECTAB=(double
*)malloc(IX1SPAN*IX2SPAN*IX3SPAN*IX4SPAN*IX5SPAN*2*sizeof(double)))==NULL){
        printf(stderr,"Error allocating sufficient memory...aborting\n");
        exit(-1);
        }
        RG1 = sqrt(G1);
        RG2 = sqrt(G2);
        RG3 = sqrt(G3);
        RG4 = sqrt(G4);
        RCZ1 = sqrt(CZ1);
        RCZ2 = sqrt(CZ2);
        RCZ3 = sqrt(CZ3);
        RCZ4 = sqrt(CZ4);
        RCC = sqrt(CC);
        for(i=0; i<LENGTH; i++)
        {
          T[i]=0.0;
          Tc[i]=0;
          Tz1[i]=0;
          Tz2[i]=0;
        }
/**** read  read datat of aggregated trajectories *********/

        trajy();

/**** initialization ****************/

        for(ix1=0; ix1<IX1SPAN; ix1++)
        {
          for(ix2=0; ix2<IX2SPAN; ix2++)
          {
              for(ix3=0; ix3<IX3SPAN; ix3++)
              {
                  for(ix4=0; ix4<IX4SPAN; ix4++)
                  {
                      for(ix5=0; ix5<IX5SPAN; ix5++)
                      {
```

```
                cotab(ix1,ix2,ix3,ix4,ix5) = 0.0;
                ectab(ix1,ix2,ix3,ix4,ix5,0) = 0.0;
                ectab(ix1,ix2,ix3,ix4,ix5,1) = 0.0;
                for(i=0; i<LENGTH; i++)
                {
                tltab(ix1,ix2,ix3,ix4,ix5,i) = 0.0;
                labtab(ix1,ix2,ix3,ix4,ix5,i,0) = Tz1[m]-3+ix1;
                labtab(ix1,ix2,ix3,ix4,ix5,i,1) = Tz1[m]-3+ix2;
                labtab(ix1,ix2,ix3,ix4,ix5,i,2) = Tz2[m]-3+ix3;
                labtab(ix1,ix2,ix3,ix4,ix5,i,3) = Tz2[m]-3+ix4;
                labtab(ix1,ix2,ix3,ix4,ix5,i,4) = Tc[m]-3+ix5;
                  }
               }
            }
         }
      }
   }
   first1=Tz1[m]+2, last1=Tz1[m]-2;
   first2=Tz2[m]+2, last2=Tz2[m]-2;
   first3=Tc[m]+2,  last3=Tc[m]-2;

while (n < count)
{
   IX1MIN = Tz1[m]-2;
   IX1MAX = Tz1[m]+2;
   IX2MIN = Tz2[m]-2;
   IX2MAX = Tz2[m]+2;
   IX3MIN = Tc[m]-2;
   IX3MAX = Tc[m]+2;
   IX1f = Tz1[m]-3;
   IX2f = Tz2[m]-3;
   IX3f = Tc[m]-3;
   for(n=m; (Tz1[m]-Tz1[n]==0 && Tz2[m]-Tz2[n]==0 && Tc[m]-Tc[n]==0
          && n<count); n++)
   ;
/***** calculate the least cost at a block ***********/

   calcu();

   if( tlu == T[m])
   {
      IX3MIN = Tc[m]-2;
      IX3MAX = Tc[m]+2;
      IX3f = Tc[m]-3;
      calcu();
   }
   m=n;
}/* end of the while loop */

printf("\nLABLE TRACES");
for(ix5=1; ix5<=IX5SPAN-2; ix5++)
{
   for(ix4=1; ix4<=IX4SPAN-2; ix4++)
   {
      for(ix3=1; ix3<=IX3SPAN-2; ix3++)
      {
        for(ix2=1; ix2<=IX2SPAN-2; ix2++)
        {
           for(ix1=1; ix1<=IX1SPAN-2; ix1++)
           {
             first1=labtab(ix1,ix2,ix3,ix4,ix5,0,0);
             last1=labtab(ix1,ix2,ix3,ix4,ix5,LM1,0);
             first2=labtab(ix1,ix2,ix3,ix4,ix5,0,1);
             last2=labtab(ix1,ix2,ix3,ix4,ix5,LM1,1);
             first3=labtab(ix1,ix2,ix3,ix4,ix5,0,2);
             last3=labtab(ix1,ix2,ix3,ix4,ix5,LM1,2);
             first4=labtab(ix1,ix2,ix3,ix4,ix5,0,3);
             last4=labtab(ix1,ix2,ix3,ix4,ix5,LM1,3);
             first5=labtab(ix1,ix2,ix3,ix4,ix5,0,4);
             last5=labtab(ix1,ix2,ix3,ix4,ix5,LM1,4);
             if(first1==last1 && first2==last2 && first3==last3 &&
             first4==last4 && first5==last5)
             {
                 for(i=0; i<LENGTH; i++)
                 {
                      printf("\nlabeltrace[%2d] = %2d,%2d,%2d,%2d,%2d",i,
                      labtab(ix1,ix2,ix3,ix4,ix5,i,0),
                      labtab(ix1,ix2,ix3,ix4,ix5,i,1),
                      labtab(ix1,ix2,ix3,ix4,ix5,i,2),
                      labtab(ix1,ix2,ix3,ix4,ix5,i,3),
                      labtab(ix1,ix2,ix3,ix4,ix5,i,4));
                      printf("\ttltrace[%2d] = %6f",i,
                      tltab(ix1,ix2,ix3,ix4,ix5,i));
                      if(tltab(ix1,ix2,ix3,ix4,ix5,i)>=0.0) break;
                     }/* end of for */
                 printf("\npeak-hour energy cost = %6f",
               ectab(ix1,ix2,ix3,ix4,ix5,1));
               printf("\noff peak-hour energy cost = %6f",
               ectab(ix1,ix2,ix3,ix4,ix5,0));

              } /* end of the if */
             }
         }
     }
  }
}
} /* end of the main function */


void calcu()
{
       double tlmax=0.0;                      /*start at midnight, go backwards (neg
time)*/
       double  tmn,time,DTMAX;
       int     tnhour,tnhour1;
       int     l, i, node1, node2, node3;


       /**** initialize the arrays *****/

       for(ix1=IX1MIN; ix1<=IX1MAX; ix1++)
       {
         for(ix2=IX1MIN; ix2<=IX1MAX; ix2++)
         {
           for(ix3=IX2MIN; ix3<=IX2MAX; ix3++)
           {
             for(ix4=IX2MIN; ix4<=IX2MAX; ix4++)
             {
              for(ix5=IX3MIN; ix5<=IX3MAX; ix5++)
              {
                 tltab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0) = T[m];
              }

             }

           }

          }

        }
       while( tlmax > T[n] )
       {

       /**** find the next node to update ****/

       tlmax = -24.0;
       for(ix1=last1+dir1; first1>last1? ix1<=first1+dir1:ix1>=first1+dir1;
       first1>last1? ix1++:ix1--)
       {
         for(ix2=last1+dir1; first1>last1? ix2<=first1+dir1:ix2>=first1+dir1;
         first1>last1? ix2++:ix2--)
         {
           for(ix3=last2+dir2; first2>last2? ix3<=first2+dir2:ix3>=first2+dir2;
           first2>last2? ix3++:ix3--)
           {
             for(ix4=last2+dir2; first2>last2? ix4<=first2+dir2:ix4>=first2+dir2;
             first2>last2? ix4++:ix4--)
             {
               for(ix5=last3+dir3; first3>last3? ix5<=first3+dir3:ix5>=first3+dir3;
              first3>last3? ix5++:ix5--)
```

```
               {
               if(tltab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0)>tlmax)
               {      ix1u=ix1;           /*node to be updated has latest last
update*/
                      ix2u=ix2;
                      ix3u=ix3;
                      ix4u=ix4;
                      ix5u=ix5;
                      tlmax=tltab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0);
                                    tlu = tlmax;
                 }

               }

              }

            }

          }

         }
        if((tlmax==T[n]))break;

        /**** find the earliest of the admissible update times*****/

        tm = 0.0;
        for(dix5=-1; dix5<=1; dix5++)
        {
          for(dix4=-1; dix4<=1; dix4++)
          {
            for(dix3=-1; dix3<=1; dix3++)
            {
              for(dix2=-1; dix2<=1; dix2++)
              {
               for(dix1=-1; dix1<=1; dix1++)
                 {
               tn=tltab(ix1n-IX1f,ix2n-IX1f,ix3n-IX2f,ix4n-IX2f,ix5n-IX3f,0);
               tmn = tn-DTMAX1;
               if(tmn<=tm) tm=tmn;
               }

              }

            }

           }

         }

        /*** find the least-cost transition ***/

        tl = tltab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0);
        ctcmin = LARGE;
        if((tl>T[n])&&(tl<=(T[n]+0.5))&&(T[m]-T[n]>0.5))
        {
               tn=tl;
               dix1=0;
               dix2=0;
               dix3=0;
               dix4=0;
               dix5=0;
               tnhour = tn+24;
               time = tn+24.0;
               tz1s=TZ1SHI;
               tz2s=TZ2SHI;
               tz3s=TZ3SHI;
               tz4s=TZ4SHI;
               if((time>8.0) && (time<=18.0))
               {
                      tz1s=TZ1SLO;
                      tz2s=TZ2SLO;
                      tz3s=TZ3SLO;
                      tz4s=TZ4SLO;
                }
               if(time<=8.0)
               price=LP;
               else
               price=HP;
               qz1 =1*qz1tab[tnhour]*3600;
               qz2 =1*qz2tab[tnhour]*3600;
               qz3 =1*qz3tab[tnhour]*3600;
               qz4 =1*qz4tab[tnhour]*3600;
               TE = tetab[tnhour];
               DTMAX=-T[n]+tn;
               dt=DTMAX;
               test1(1);

        }
        else
        {
               for(dix5=-1; dix5<=1; dix5++)
               {
                 for(dix4=-1; dix4<=1; dix4++)
                 {
                   for(dix3=-1; dix3<=1; dix3++)
                   {
                     for(dix2=-1; dix2<=1; dix2++)
                     {
                       for(dix1=-1; dix1<=1; dix1++)
                      {
                             tn = t1tab(ix1n-IX1f,ix2n-IX1f,ix3n-IX2f,ix4n-IX2f,ix5n-
IX3f,0);
                             tnhour = tn+24;
                             time = tn+24.0;
                             qz1 =1*qz1tab[tnhour]*3600;
                             qz2 =1*qz2tab[tnhour]*3600;
                             qz3 =1*qz3tab[tnhour]*3600;
                             qz4 =1*qz4tab[tnhour]*3600;
                             TE = tetab[tnhour];
                             tz1s=TZ1SHI;
                             tz2s=TZ2SHI;
                             tz3s=TZ3SHI;
                             tz4s=TZ4SHI;
                             if((time>8.0) && (time<=18.0))
                             {
                             tz1s=TZ1SLO;
                             tz2s=TZ2SLO;
                             tz3s=TZ3SLO;
                             tz4s=TZ4SLO;
                             }
                             if(time<=8.0)
                             price=LP;
                             else
                             price=HP;
                             if((n-m==1)||(T[m]-T[n]<=0.5)){
                             DTMAX=tn-T[n];
                             for(l=1; l>=1; l--)
                             {
                             dt=DTMAX/l;
                             if((time>=tnhour) && (tnhour>(time-dt)))
                             {
                             tnhour1=24+tn-dt;
                             qz1=1*(qz1tab[tnhour1]*(tnhour-(time-
dt))/dt+1*qz1tab[tnhour]
                                *(time-tnhour)/dt)*3600;

                             qz2=1*(qz2tab[tnhour1]*(tnhour-(time-
dt))/dt+1*qz2tab[tnhour]
                                *(time-tnhour)/dt)*3600;
                             qz3=1*(qz3tab[tnhour1]*(tnhour-(time-
dt))/dt+1*qz3tab[tnhour]
                                *(time-tnhour)/dt)*3600;
                             qz4=1*(qz4tab[tnhour1]*(tnhour-(time-
dt))/dt+1*qz4tab[tnhour]
                                *(time-tnhour)/dt)*3600;
                             TE=(tetab[tnhour1]*(tnhour-(time-
dt))/dt+tetab[tnhour]*(time-tnhour)/dt);

                             }/* end of if */
                             test1(1);
```

```
                              }/* end of for */
                      }
                      else
                      {
                              DTMAX=DTMAX1;
                              for(l=LMAX; l>=1; l--)
                              {
                              dt=DTMAX/l;
                              if((time>=tnhour) && (tnhour>(time-dt)))
                              {
                               tnhour1=24+tn-dt;
                               qz1=1*(qz1tab[tnhour1]*(tnhour-(time-
dt))/dt+1*qz1tab[tnhour]
                                 *(time-tnhour)/dt)*3600;

                               qz2=1*(qz2tab[tnhour1]*(tnhour-(time-
dt))/dt+1*qz2tab[tnhour]
                                 *(time-tnhour)/dt)*3600;
                               qz3=1*(qz3tab[tnhour1]*(tnhour-(time-
dt))/dt+1*qz3tab[tnhour]
                                 *(time-tnhour)/dt)*3600;
                               qz4=1*(qz4tab[tnhour1]*(tnhour-(time-
dt))/dt+1*qz4tab[tnhour]
                                 *(time-tnhour)/dt)*3600;
                               TE=(tetab[tnhour1]*(tnhour-(time-
dt))/dt+tetab[tnhour]*(time-tnhour)/dt);

                              }/* end of if */
                              test1(l);
                              }/* end of for */
                      }/* end of else */
                     }

                   }

                   }

               }

               }
              }/* end of the else */
              if(tlu >= tlmax)
              {
               for(ix1=IX1MIN; ix1<=IX1MAX; ix1++)
               {
                 for(ix2=IX1MIN; ix2<=IX1MAX; ix2++)
                 {
                   for(ix3=IX2MIN; ix3<=IX2MAX; ix3++)
                   {
                     for(ix4=IX2MIN; ix4<=IX2MAX; ix4++)
                     {
                       for(ix5=IX3MIN; ix5<=IX3MAX+1; ix5++)
                       {
                      cotab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3)
                      = cotab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f);
                      ectab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3,1)
                      = ectab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,1);
                      ectab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3,0)
                      = ectab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0);
                       for(i=0; i<LENGTH; i++)
                      {
                      tltab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3,i)
                       = tltab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i);
                       labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3,i,0)
                      = labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,0);
                       labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3,i,1)
                      = labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,1);
                       labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3,i,2)
                      = labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,2);
                       labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3,i,3)
                      = labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,3);
                       labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f+dir3,i,4)
                      = labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,4);
                      }
                      }
```

```
                    }
                  }
                }
              }
              for(i=m+1; Tc[m]-Tc[i]>=0; i++)
             {
                 Tc[i]=Tc[i]+1;
             }
             Tc[m]=Tc[m]+1;
             dir3=0;
             return;
        }/* end of the if */

        for(i=LENGTH-2; i>=0; i--)
        {
               tltab(ixlu-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,i+1) =
                tltab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-IX2f
               ,ix5next-IX3f,i);
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,i+1,0)
               = labtab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-IX2f
               ,ix5next-IX3f,i,0);
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,i+1,1)
               = labtab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-IX2f
               ,ix5next-IX3f,i,1);
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,i+1,2)
               = labtab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-IX2f
               ,ix5next-IX3f,i,2);
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,i+1,3)
               = labtab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-IX2f
               ,ix5next-IX3f,i,3);
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,i+1,4)
               = labtab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-IX2f
               ,ix5next-IX3f,i,4);
               }
               cotab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f) = nco;
               if (tlu<-16.0)
               {
                  ectab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0) =
                   nec+ectab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-
IX2f,ix5next-IX3f,0);
                  ectab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,1) =
                   ectab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-IX2f,ix5next-
IX3f,1);
               }
               else
               {
                  ectab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,1) =
                   nec+ectab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-
IX2f,ix5next-IX3f,1);
                  ectab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0) =
                   ectab(ix1next-IX1f,ix2next-IX1f,ix3next-IX2f,ix4next-IX2f,ix5next-
IX3f,0);
               }
               tltab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0)= tlu;
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0,0) = ix1u;
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0,1) = ix2u;
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0,2) = ix3u;
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0,3) = ix4u;
               labtab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f,0,4) = ix5u;

        } /*** closes the "while" loop in main() ***/
                 if((Tz1[m]-Tz1[n])>0)
                 {
                   dir1=-1;
                   IX1l=IX1f-1;
                   first1=IX1MAX;
                   last1=IX1MIN;
                 }
                 else if((Tz1[m]-Tz1[n])<0)
                 {
                   dir1=1;
                   IX1l=IX1f+1;
                   first1=IX1MIN;
                   last1= IX1MAX;
                   }
                  else
                  {
                    dir1=0;
                    IX11=IX1f;
                    first1=IX1MAX;
                    last1=IX1MIN;
                    }
                  if((Tz2[m]-Tz2[n])>0)
                  {
                    dir2=-1;
                    IX21=IX2f-1;
                    first2=IX2MAX;
                    last2=IX2MIN;
                  }
                  else if((Tz2[m]-Tz2[n])<0)
                  {
                    dir2=1;
                    IX21=IX2f+1;
                    first2=IX2MIN;
                    last2= IX2MAX;
                    }
                  else
                  {
                    dir2=0;
                    IX21=IX2f;
                    first2=IX2MAX;
                    last2=IX2MIN;
                    }
                  if((Tc[m]-Tc[n])>0)
                  {
                    dir3=-1;
                    IX31=IX3f-1;
                    first3=IX3MAX;
                    last3=IX3MIN;
                  }
                  else if((Tc[m]-Tc[n])<0)
                  {
                    dir3=1;
                    IX31=IX3f+1;
                    first3=IX3MIN;
                    last3= IX3MAX;
                    }
                  else
                  {
                    dir3=0;
                    IX31=IX3f;
                    first3=IX3MAX;
                    last3=IX3MIN;
                    }
        for(ix1=first1;  first1>last1? ix1>=last1:ix1<=last1; first1>last1? ix1--
:ix1++)
        {
             for(ix2=first1;  first1>last1? ix2>=last1:ix2<=last1; first1>last1? ix2--
:ix2++)
             {
                for(ix3=first2; first2>last2? ix3>=last2:ix3<=last2; first2>last2? ix3-
-:ix3++)
                {
                    for(ix4=first2; first2>last2? ix4>=last2:ix4<=last2; first2>last2?
ix4--:ix4++)
                    {
                        for(ix5=first3; first3>last3? ix5>=last3:ix5<=last3;
first3>last3? ix5--:ix5++)
                        {
                        cotab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31) =
                           cotab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f);
                         ectab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,1) =
                           ectab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,1);
                         ectab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,0) =
                          ectab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0);
                         tltab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,0)=
                           tltab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0);
                         labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,0,0)=
                        labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0,0);
                         labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,0,1)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0,1);
                          labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,0,2)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0,2);
                          labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,0,3)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0,3);
                          labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,0,4)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,0,4);
                            for(i=1; i<LENGTH; i++)
                         {
                         tltab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,i)=
                          tltab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i);
                          labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,i,0)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,0);
                          labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,i,1)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,1);
                          labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,i,2)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,2);
                          labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,i,3)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,3);
                          labtab(ix1-IX11,ix2-IX11,ix3-IX21,ix4-IX21,ix5-IX31,i,4)=
                         labtab(ix1-IX1f,ix2-IX1f,ix3-IX2f,ix4-IX2f,ix5-IX3f,i,4);

                           }/* end of for LENGTH */
                         }
                          }
                       }
                     }
                  }
}/* end of the  function */

/*********functions to compute new-node index*************/

int fix1n()
{
        extern int ix1u, dix1;
        int i;
        if(tl==T[m]&&(m!=0)&&dirl==-1)
        {
            if((ix1u==(IX1MIN+1) && dix1<0)||(ix1u==IX1MAX && dix1>0)) i=0;
            else if(ix1u==IX1MIN) i=1;
            else i=dix1;
         }
        else if(tl==T[m]&&(m!=0)&&dirl==1)
        {
            if((ix1u==(IX1MAX-1) && dix1>0)||(ix1u==IX1MIN && dix1<0)) i=0;
            else if(ix1u==IX1MAX) i=-1;
            else i=dix1;
         }
        else
        {
          if((ix1u==IX1MIN && dix1<0)||(ix1u==IX1MAX && dix1>0)) i=0;
          else i=dix1;
         }
        return(ix1u+i);
}
int fix2n()
{
        extern int ix2u, dix2;
        int j;
        if(tl==T[m]&&(m!=0)&&dirl==-1)
        {
            if((ix2u==(IX1MIN+1) && dix2<0)||(ix2u==IX1MAX && dix2>0)) j=0;
            else if(ix2u==IX1MIN) j=1;
            else j=dix2;
         }
        else if(tl==T[m]&&(m!=0)&&dirl==1)
        {
            if((ix2u==(IX1MAX-1) && dix2>0)||(ix2u==IX1MIN && dix2<0)) j=0;
            else if(ix2u==IX1MAX) j=-1;
            else j=dix2;
         }
        else
        {
          if((ix2u==IX1MIN && dix2<0)||(ix2u==IX1MAX && dix2>0)) j=0;
          else j=dix2;
         }
        return(ix2u+j);
}

int fix3n()
{
        extern int ix3u, dix3;
        int k;
        if(tl==T[m]&&(m!=0)&&dir2==-1)
        {
            if((ix3u==(IX2MIN+1) && dix3<0)||(ix3u==IX2MAX && dix3>0)) k=0;
            else if(ix3u==IX2MIN) k=1;
            else k=dix3;
         }
        else if(tl==T[m]&&(m!=0)&&dir2==1)
        {
            if((ix3u==(IX2MAX-1) && dix3>0)||(ix3u==IX2MIN && dix3<0)) k=0;
            else if(ix3u==IX2MAX) k=-1;
            else k=dix3;
         }
        else
        {
          if((ix3u==IX2MIN && dix3<0)||(ix3u==IX2MAX && dix3>0)) k=0;
          else k=dix3;
         }
        return(ix3u+k);
}

int fix4n()
{
        extern int ix4u, dix4;
        int l;
        if(tl==T[m]&&(m!=0)&&dir2==-1)
        {
            if((ix4u==(IX2MIN+1) && dix4<0)||(ix4u==IX2MAX && dix4>0)) l=0;
            else if(ix4u==IX2MIN) l=1;
            else l=dix4;
         }
        else if(tl==T[m]&&(m!=0)&&dir2==1)
        {
            if((ix4u==(IX2MAX-1) && dix4>0)||(ix4u==IX2MIN && dix4<0)) l=0;
            else if(ix4u==IX2MAX) l=-1;
            else l=dix4;
         }
        else
        {
          if((ix4u==IX2MIN && dix4<0)||(ix4u==IX2MAX && dix4>0)) l=0;
          else l=dix4;
         }
        return(ix4u+l);
}

int fix5n()
{
        extern int ix5u, dix5;
        int o;
        if(tl==T[m]&&(m!=0)&&dir3==-1)
        {
            if((ix5u==(IX3MIN+1) && dix5<0)||(ix5u==IX3MAX && dix5>0)) o=0;
            else if(ix5u==IX3MIN) o=1;
            else o=dix5;
         }
        else if(tl==T[m]&&(m!=0)&&dir3==1)
        {
            if((ix5u==(IX3MAX-1) && dix5>0)||(ix5u==IX3MIN && dix5<0)) o=0;
            else if(ix5u==IX3MAX) o=-1;
            else o=dix5;
         }
        else
        {
           if((ix5u==IX3MIN && dix5<0)||(ix5u==IX3MAX && dix5>0)) o=0;
           else o=dix5;
         }
        return(ix5u+o);
}
```

```
/************* procedure to call trial cost computation *****/

void testl(int l)
{
        if (dt==0.0) return;
        v1 = (ix1n-ix1u)*DX1/dt - d1 + x1u*AZ1/CZ1;
        v2 = (ix2n-ix2u)*DX2/dt - d2 + x2u*AZ2/CZ2;
        v3 = (ix3n-ix3u)*DX3/dt - d3 + x3u*AZ3/CZ3;
        v4 = (ix4n-ix4u)*DX4/dt - d4 + x4u*AZ4/CZ4;
        v5 = (ix5n-ix5u)*DX5/dt - d5 + x5u*AC/CC;
        u1 = v1/(B1*g1);
        u2 = v2/(B2*g2);
        u3 = v3/(B3*g3);
        u4 = v4/(B4*g4);
        u5 = (RG1*v1 + RG2*v2 + RG3*v3 + RG4*v4 + v5)/(C55 + B5*g5);
        if((u1>=0.0) && (u2>=0.0) && (u3>=0.0) && (u4>=0.0) && (u5>=0.0))
        {
                test();
        }
                return;
}


/*****  procedure to compute trial cost *****/

void test()
{
        double tc, ct, ctc,tltry;
        tltry=tn-dt;
        if((T[m]-T[n]>0.5)&&(tltry<T[n])&&(tl>(T[n]+0.5)))return;
        tc = ((R1*u1*u1 + R2*u2*u2 + R3*u3*u3 + R4*u4*u4 + u5 )*price+
                  A1*pow(x1u-X1S,2.0)/CZ1 +
                  A2*pow(x2u-X2S,2.0)/CZ2 +
                  A3*pow(x3u-X3S,2.0)/CZ3 +
                 A4*pow(x4u-X4S,2.0)/CZ4)*dt +
                   cotab(ix1n-IX1f,ix2n-IX1f,ix3n-IX2f,ix4n-IX2f,ix5n-IX3f);
        ct = (tc - cotab(ix1u-IX1f,ix2u-IX1f,ix3u-IX2f,ix4u-IX2f,ix5u-IX3f))
              * (tn-dt-tm)/(tl-(tn-dt));
        ctc = tc + ct;
        if(ctc<=ctcmin)
        {
                ctcmin=ctc;
                nco=tc;
                tlu=tn-dt;
                nec=(R1*u1*u1 + R2*u2*u2 + R3*u3*u3 + R4*u4*u4 + u5)*dt ;
                ix1next = ix1n;
                ix2next = ix2n;
                ix3next = ix3n;
                ix4next = ix4n;
                ix5next = ix5n;
        }
        return;
}


/***************read oringnal result from input file*****************/
void trajy()
{
        int j;
         FILE *fp;
        fp=fopen("input","r");
         fscanf(fp,"%3d",&count);
        for(j=0; j<=count; j++)
        {
                fscanf(fp,"%2d",&Tz1[j]);
                fscanf(fp,"%2d",&Tz2[j]);
                fscanf(fp,"%2d",&Tc[j]);
                fscanf(fp,"%lf",&T[j]);
         }
        fclose(fp);
        return;
```